BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

UNITED STATES PROVISIONAL PATENT APPLICATION

FOR

KINZAN SYSTEM

Inventors:
Dyami Calire
Carlos Chue
Reza Ghanbari
Anthony Tang
Eric VanLydegraf

Prepared by:
BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP
12400 Wilshire Boulevard
Seventh Floor
Los Angeles, California 90025-1026
(408) 720-8598

Attorney's Docket No: 004348.P005Z

"Express Mail" mailing label number: EL 639 015 475 US
Date of Deposit: August 25, 2000
I hereby certify that I am causing this paper or fee to be deposited with the United States Postal Service "Express Mail Post Office to Addressee" service on the date indicated above and that this paper or fee has been addressed to the Assistant Commissioner for Patents, Washington, D. C. 20231

Glenn E. Von Tersch
(Typed or printed name of person mailing paper or fee)

Glenn E. Von Tersch
(Signature of person mailing paper or fee)

August 25 2000
(Date signed)

KINZAN SYSTEM

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all copyright rights whatsoever.

In the following description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the invention. It will be apparent, however, to one skilled in the art that the invention can be practiced without these specific details. In other instances, structures and devices are shown in block diagram form in order to avoid obscuring the invention.

Reference in the specification to "one embodiment" or "an embodiment" means that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one embodiment of the invention. The appearances of the phrase "in one embodiment" in various places in the specification are not necessarily all referring to the same embodiment, nor are separate or alternative embodiments mutually exclusive of other embodiments.

Kinzan Technology Platform Framework Overview

1 Introduction

The Fountainhead release of the Kinzan Technology Platform (KTP) allows for easy extensibility and assembly of web-based applications by developers and development partners with a wide variety of skills.

The various components of the KTP Framework may help untangle the stew of static and active content, style, structure, application logic, templates, servlets, and persistence mechanisms that is the norm in Java Server Pages (JSP) driven application environments. The result represents a major commitment to flexibility, configurability, extensibility, integrability, scalability, and reliability at all levels of application development.

1.1 Assembly Is Easier Than Building

The KTP Framework may enable the software engineer to work collaboratively with information architects, web developers, graphics designers, and customer advocates to create compelling network-based applications that can be delivered to different types of network devices (i.e., web phones, browsers, etc.) and with different branding and localization to different languages.

The KTP Framework offers a series of abstractions that modularize and generalize application development at all levels of the application architecture. The result is an environment that enables modular assembly of applications from pre-built modules, while minimizing the amount of custom work and maintenance required for different customers.

In one embodiment, at the front end, the primary mechanism for assembling a web page is a Kinzan Server Page (KSP). A KSP is actually a collection of configuration and content files. The rendering process involves a recursive assembly and transformation of a given page into a series of precompiled Java servlet modules, while allowing developers to collaboratively develop and maintain modules at a level that is appropriate to their role and skill set.

Similarly, in one embodiment, at the back end, the Kinzan Services Manager provides a powerful and flexible mechanism for integrating and coordinating a large variety of back-end databases, information sources, enterprise applications, and services.

Moreover, in one embodiment, bridging the front and back ends is an extremely rich environment for assembling application logic and flow from application modules, supported by the Kinzan State Manager.

1.2 Make World Class Technologies Accessible

In one embodiment, at all levels the KTP Framework leverages the best the Java 2 Enterprise Edition (J2EE) platform and XML-based technologies have to offer, and takes it to a new level. The result is a highly scalable, reliable, extensible, and distributable platform, radically decreasing development time.

By modularizing and generalizing much of server-based application development, the KTP Framework facilitates development of complex enterprise-hosted applications, enabling fully branded, highly integrated customer experiences.

Perhaps more importantly, the modularization and significant simplification of the core J2EE technologies allows highly productive and impactful contribution by development team members with a wide variety of technical and design skills. Each can focus on modules where their skills have the greatest impact. Extensive modularization also encourages reuse and sharing of modules, increasing productivity and quality in the process.

1.3 End To End Support

In one embodiment, the KTP Framework is more than just software. It allows seamless integration with a high-availability, high-scalability application hosting environment. The result is a technology platform that enables developers to rapidly develop compelling applications, and rapidly and painless deploy them to a mission-critical hosting environment that can grow as demand for their applications grow.

1.4 One Platform, One Web

In one embodiment, the KTP Framework has been built to bridge a myriad of back end services and data sources to multiple target devices (web phones, browsers, PDAs, etc.) with multiple branding and potentially to multiple languages and locale.

Services that are integrated on the back end are made available to application developers as Kinzan Widgets. Widgets are able to support unique presentation and branding to multiple target devices and locale, enabling the web developer to take advantage of the rich functionality the widget is exposing.

As Kinzan and its partners integrate services such as e-mail and payment processing in the KTP Framework, all applications built with the KTP Framework have the functionality available to drop in. The end result is a progressively growing set of services and widget components that are available to be assembled into compelling branded applications.

2 Motivation

In one embodiment, the KTP Framework is intended to enable rapid branding, assembly, extension, and configuration of generic application modules. It does so by modularizing style, presentation, content, application logic, business logic, and services, and providing runtime capabilities that dynamically assemble these modules into rich applications.

The result is a development and deployment environment that cleanly separates the various elements required to build dynamic web applications. By separating these elements, teams can work more effectively, with various team members contributing to modules that require their skill sets, rather than requiring all team members to have multiple skill sets (i.e., graphics design, page layout, Java programming, etc.).

2.1 Market Drivers

Current server-based application development technologies typically tightly couple presentation, content, and logic, making modular development of brandable applications impractical. Each application may be a one-time implementation, compounding support and maintenance issues going forward.

Kinzan has a long history of delivering uniquely branded and configured Internet content and commerce management systems to our customers on our hosted platform. The result is a deep understanding of application frameworks that enable rapid branding and configuring of generic applications.

Competitive pressures mean time to value delivery is helpful. This places a premium on being able to rapidly and easily assemble, reconfigure, and extend generic applications. The global nature of the Internet also places a premium on easy localization and support for emerging alternative devices for Internet access (PDAs, phones, pagers, etc.).

Finally, supporting an application development and deployment environment in a shared hosting facility may include providing robust security and reliability to support multiple development partners simultaneously.

2.2 Technology Drivers

Even the most sophisticated technologies have limited value if they are too complex to be used. Thus it may be advantageous to develop and package the most sophisticated technologies in a way that empowers teams with varying technical skills.

Typical client-server implementations separate the client from the database. In the JSP world, that means direct connectivity between the JSP and the database via Java Database Connectivity (JDBC) queries (so-called Java Model 1 architecture).

With Enterprise Java Beans (EJBs), developers may abstract out business logic into a separate services layer using entity EJBs and session EJBs. However, that still leaves presentation and application logic in the JSP layer. The use of tag libraries with JSP 1.1 helps extract application and presentation logic associated with common components out of the JSP, but the different elements are still co-mingled. Combining JSPs and servlets in a Model-View-Controller (MVC) configuration (so-called Java Model 2 architecture) alleviates some issues, but generally requires a higher level of sophistication for all developers on a project.

In one embodiment, the KTP Framework (including rendering, state management, and services management processes) cleanly separates style, layout, presentation, and content from application and business logic, with capability of supporting localization and output to target deployment platforms with different capabilities (PDAs, browsers, pagers, etc.).

Similarly, in one embodiment, the collection of files that make up a KSP are the "source code" that is "compiled" into a series of modular servlets (and associated data), which are then dynamically assembled by the rendering engine at runtime. In this way, developers may take full advantage of the power and performance of Java servlets, while benefiting from increased reuse and modularity and the clean separation of function and responsibilities in the development process.

Moreover, in one embodiment, the Kinzan State Manager uses XML-based wizards to connect and configure application logic modules, effectively assembling the controller tier of the application.

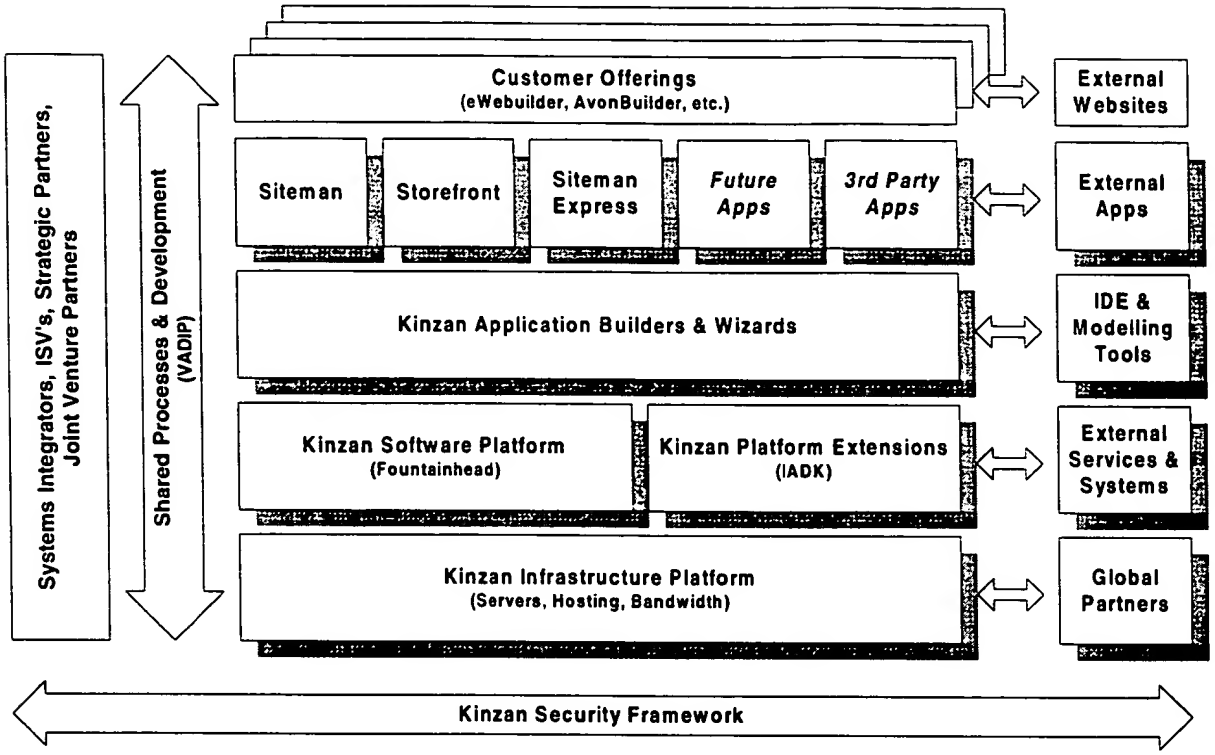
Furthermore, in one embodiment, the Kinzan Services Manager provides an abstraction layer for many kinds of backend services and includes the ability to dynamically look up to and bind to services. The result is modularization of the model tier.

Likewise, in one embodiment, the KTP Framework is flexible. If developers choose to implement their pages "close to the metal" as standard JSP pages or Java servlets, they may. However, do so is to give up the benefits of being able to apply different style and branding elements to a given application. In practice, developers may leverage the KTP Framework for elements of their applications that require flexible styling and branding, while applying conventional HTML, XML, and JSP techniques for fragments where the flexibility is not required.

Also, in one embodiment, the KTP Framework is designed to support the development and deployment of closely related applications across multiple capability devices (pagers, PDAs, etc.) and locales (English, French, etc.).

As a practical matter, it is unlikely that there will ever be a painless way to simultaneously deploy a single dynamic web application to standard web browsers and a two-way pager with 20-character display, no matter how sophisticated a system. However, embodiments of the KTP Framework allow developers to rapidly develop and deploy closely related applications to platforms with different capabilities. By only needing to change the presentation modules, developers can leverage a common development methodology, business logic, localization, and data persistence layers between all apps.

3 Kinzan Technology Platform Overview



In one embodiment, the Kinzan Technology Platform (KTP) is composed of several layers as described below.

At the base is the Kinzan Infrastructure Platform. This represents the high-availability, high-scalability hosting infrastructure for the KTP. These data centers can be shared among many partners and customers, giving each access to more capacity and more reliability at less cost.

The Kinzan Software Platform (also know as Fountainhead) represents the core services and capabilities described in this document. It is a shared foundation for all applications built on top of the KTP.

The core platform can also be extended by integrating in new services, then exposing these services with widgets and wizards.

The modular nature of the KTP encourages reuse at all levels. Collections of modules (widgets, wizards, and support modules) form libraries that may be used to streamline application development.

Kinzan's own generic content management and commerce management applications may be assembled from these modules. In turn, these generic capabilities may be assembled and configured into uniquely branded offerings for each customer.

Kinzan's Virtual Application Development and Integration Process (VADIP) can tie all these pieces together, while the Kinzan Security Framework may offer enhanced information security above and beyond what is typically found in the Java platform.

3.1 Empowering Project Teams

In one embodiment, at all levels the KTP Framework works to make technology accessible to teams with different skill levels.

Specifically, it provides a full dynamic Rendering Service that assembles various applications on the fly. A benefit of this approach is the ability for applications to directly configure runtime environment of other applications, effectively resulting in codeless development for those interested in maintaining their own applications.

For web developers, the KTP Framework exposes powerful J2EE features as HTML-like tags for advanced components (Kinzan Widgets), allowing web developers to focus on creating new and better presentations for the generic functionality in these widgets.

For graphics designers, the KTP Framework supports a Cascading Style Sheets (CSS) -like approach to styling generated output. However, unlike traditional CSS (which requires that the browser client interpret the CSS), the KTP Framework applies the CSS transformations on the server-side. The result is nearly all of the benefit of CSS, without the client compatibility issues.

In addition, the KTP Framework allows for easy management of assets (images, documents, etc.), generically managing asset variants based on any combination of style, locale, and/or target device. For example, a logo asset may have a large JPEG variant for web pages, a small BMP variant for web phones, multiple color variants to compliment different corporate color schemes, and perhaps a localized slogan for different regions of the world.

In one embodiment, the graphics designer would manage all these variants, and the rest of the team would automatically inherit these variants whenever they use the logo asset. As the asset is enriched with new variants, the Kinzan Rendering Service will always resolve the variant that is most appropriate to the user that is requesting a page from the web application.

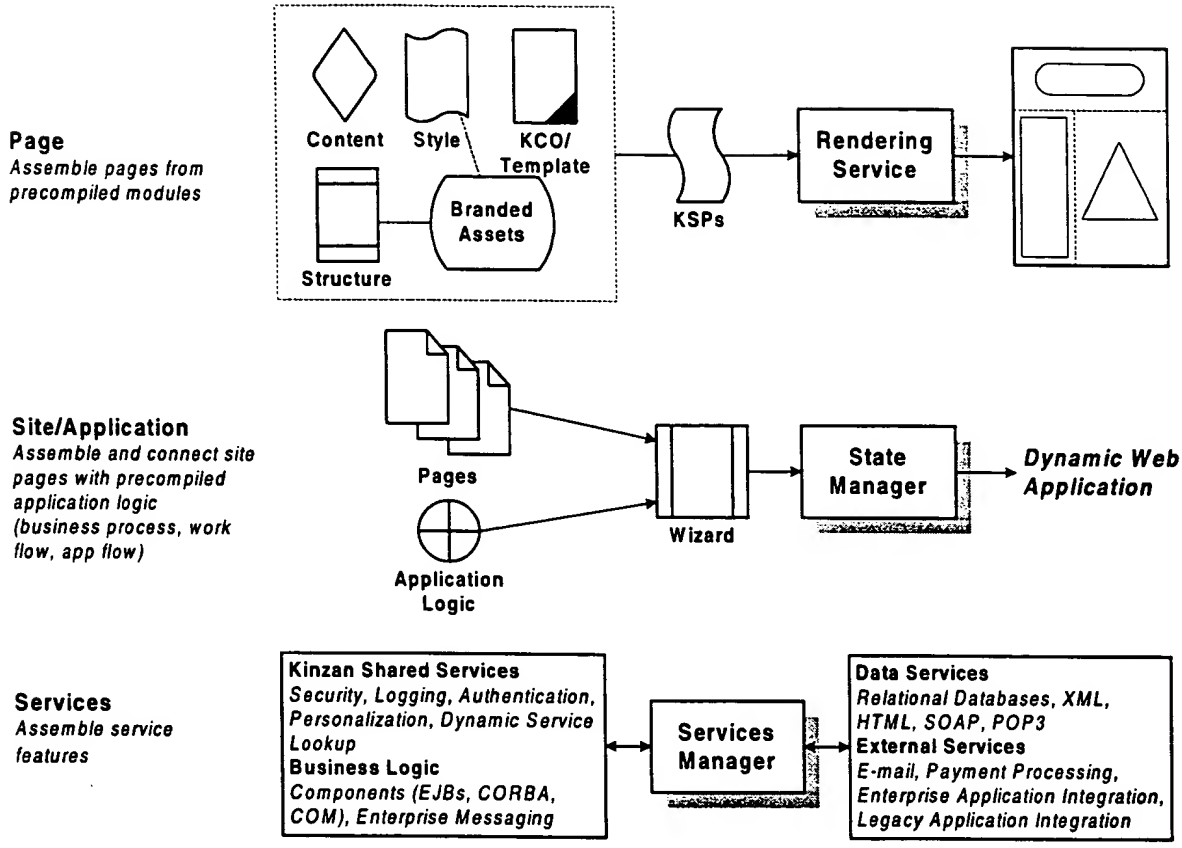
For more advanced web developers, the KTP Framework supports an enhanced version of Java Server Pages that is used to define in-line logic and presentation for components (widgets). Presentation for widgets is managed as an asset, and thus may have multiple variants based on style, locale, and/or target device. Managing presentation for widgets is the bridge between web development and software engineering, and leverages the strengths of web developers who have acquired some sophistication with JSP programming.

For mid-level software engineers, the widget component model for presentation and the pipeline component model for application logic enable development of cleanly encapsulated components that can be easily configured and connected together using XML.

For more advanced software engineers, the Kinzan Services Manager enables sophisticated integration and development of back-end services and components, including Enterprise Java Beans for business logic and integration with existing enterprise systems. These services may then be exposed with widgets and rolled out to many applications.

At all levels of the KTP Framework, support for dynamic assembly of modules enables easy reuse of modules with resource libraries (pages, components, styles, structures, etc.). The result is more effective development, less maintenance and support, and higher quality systems.

3.2 Modular Assembly Approach



In one embodiment, the KSP Framework is based on a modular assembly approach to page and application definition. By leveraging real-time modular assembly of pages by the Rendering System, and modular assembly and control of applications by the Kinzan State Manager, KTP-based applications are intrinsically extremely flexible and configurable.

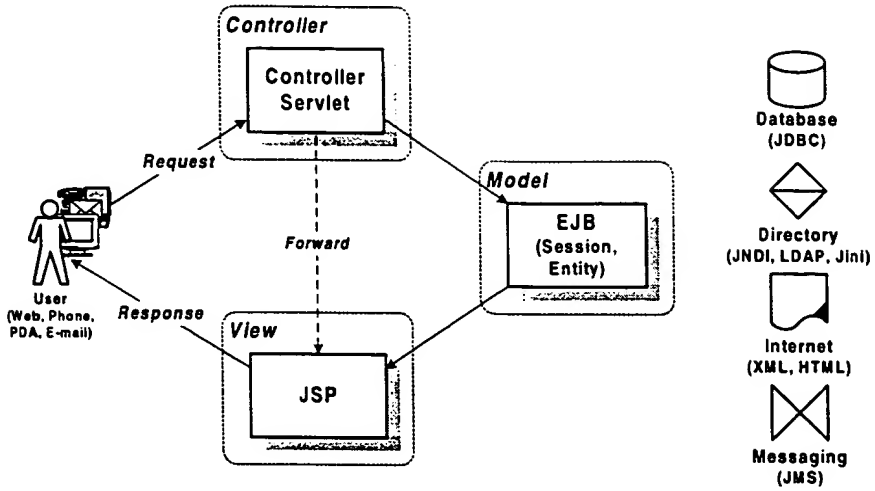
The modular assembly approach also empowers less technically sophisticated team members to assemble sophisticated applications, rather than having to code the various elements themselves. More importantly, it makes it practical to develop tools and applications that directly manipulate and configure other applications, empowering customers to maintain their own applications via "codeless" development.

In one embodiment, the KSP Framework decomposes style, structure, presentation, active content, branded assets, application logic, business logic, and various support services into distinct modules. Development team members contribute by developing modules that are appropriate to their skill set. These modules are then assembled to describe web pages and web applications.

The KSP is the artifact that describes the modules to be used to assemble a page, and the Wizard is the artifact that describes how pages and application logic are to be assembled into dynamic web applications.

Although some embodiments of the KTP Framework attempt to isolate dependencies between the various element types described hereafter, there are situations where rules are necessary to control the coupling between element types. Examples are included in the next section. In these cases, the KTP Framework allows definition of module variants that are bound to any combination of style, locale, and/or target device. Changing any of these instantly changes the module variant that is used when rendering a page or driving the application.

3.3 Request-Response Architecture

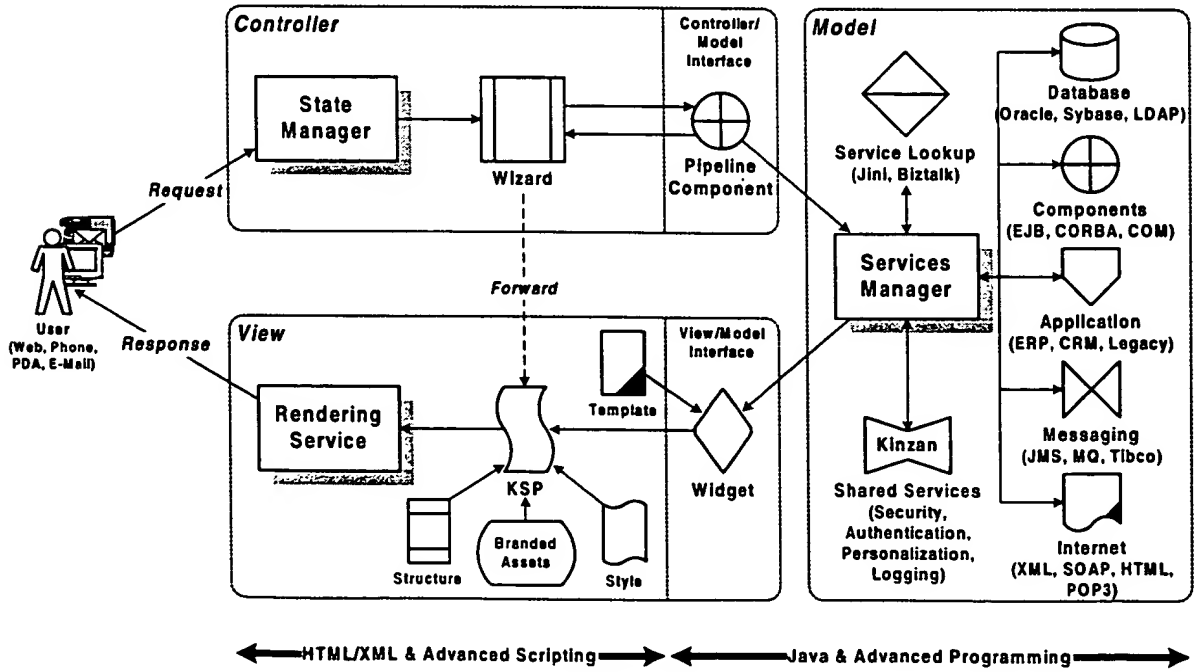


Request-Response: Java Model 2 Architecture

In the Java Model 2 architecture, Java Server Pages (JSPs) are used to query Enterprise Java Beans (EJBs) to generate a dynamic web page for a user. Different JSPs may be written to output different XML variants to support different kinds of devices.

HTML is embedded in the JSP, along with the Java programming required to access the EJBs. Since sophisticated developers and sophisticated web designers are rarely the same person, it is often difficult to support a collaborative development environment.

When a user performs an action (say, clicking the check out button at an e-commerce site), the web page passes the action to a Java controller servlet. This servlet processes the action, stores information in EJBs if necessary, then asks that the next appropriate JSP get rendered.



Request-Response: Kinzan T chn l gy Platform

In one embodiment, the KTP Framework takes each of these layers and modularizes it. At the view layer, the Rendering Service dynamically assembles pages from all the modules that make up the page. Interfacing with the model layer is managed through widgets, minimizing the impact on those that are more expert with graphics design and page presentation.

The State Manager performs a similar function at the controller layer. Wizards are used to dynamically assemble and configure application logic modules call pipeline components. Interfacing with the model layer is managed with pipeline components, allowing information architects and site designers to more easily manage configuring application flow.

The Services Manager provides an abstraction and resolution layer on top of many kinds of services and components, making it easier for widget and application logic developers to take advantage of rich backend services at the model layer.

3.3.1 View Layer

In contrast to the use of monolithic Java Server Pages (JSPs) in a typical Java Model 2 architecture, some embodiments of KSPs allow the easy modularization and dynamic assembly of structure, style, dynamic presentation, and branded assets. Since pages are dynamically assembled and rendered by the Rendering Service, KSPs allow for extremely easy and rapid deployment of pages to different target devices (wireless phones, PDAs, web browsers, etc.) with different styles and brand and across different locales.

In some embodiments of the KTP architecture, interfacing between the view and model layers (typically the most technically sophisticated and daunting component of a multi-tier architecture for non-programmers) is consolidated and encapsulated into easily reusable widgets.

Presentation for widgets may be managed external to the widget with templates This allows graphics designers, information architects, and web developers to focus on creating compelling customer experiences, and not the nitty gritty of multi-tier application development.

For widgets that are intended to expose dynamic content, the widget definition and the template may be sufficient. Widgets that are intended to drive application functionality may require collaboration with the Kinzan State Manager for the controller tier. These widgets typically have their own wizards and pipeline components, although they may certainly work with other wizards and pipeline components.

Widgets may also contain other widgets, allowing developers to present components at the appropriate level of granularity to page designers. For instance, a shopping basket widget may include a couple address widgets, several line item widgets, and a shipping and tax calculator widget, for example. In this way, page designers have the flexibility to either use an off-the-shelf shopping basket widget, or ask that a more specialized shopping basket widget be made by assembling and rearranging the more atomic widgets in a different way.

As with all the various other modular artifacts in the KTP Framework, widgets can be inherited and shared between different sites and applications, encouraging true reuse and consistency when assembling (versus building) applications.

In some embodiments, a KSP is actually a collection of references to modules representing style, structure, presentation, content, and branded assets. Page configuration and assembly instructions are captured in the KSP file as XML, and used to intelligently drive the rendering process.

When a page is requested, the Rendering Service uses the page description in the KSP to recursively assembles the various modules that make up the page. To be available to a page, the modules need to have been compiled and loaded into the Rendering Service as a compiled Java servlet. Modules that are loaded into the Rendering Service are available to be used by any KSP associated with the site.

The KSP is primarily a development aid to describe the various modules that collectively make up a web page. A KSP Processor is used to intelligently compile the KSP into a series of JSP files that are compiled and executed as a Java servlets by the Rendering Service. Developers are encouraged to use

generic tags for presentation, leaving final transformation to HTML as a terminal transformation step by the Rendering Service.

Modules may be localized and/or configured for different capability devices (phone, pagers, etc.). During the rendering process, specific module variants may be processed or assembled in response to external target variables. Example variables are target language (e.g., EN, FR, etc.), target device type (e.g., HTML, WML, PDA), and a reference to a template bundle that contains application wide or site wide resources, and configuration information.

3.3.2 Controller Layer

In contrast to the use of monolithic (and often custom to each section of every application) controller servlets in a typical Java Model 2 architecture, wizards may allow the easy modularization and dynamic assembly of application login and transitions between different states of an application.

In some embodiments of the KTP architecture, interfacing between the controller and model layers (typically a very daunting component of a multi-tier architecture for non-programmers) is consolidated into easily reusable pipeline components. Widgets may be closely associated with their own wizards, effectively preconfiguring the controller layer for web developers. This allows web developers to focus on creating compelling customer experiences and not the nitty gritty of multi-tier application development.

Preferably, all the various modular artifacts in the KSP framework may be inherited and shared between different sites and applications, encouraging true reuse and consistency when assembling (versus building) applications.

If the functionality of the preconfigured wizards needs to be adjusted, the various components of the wizard definition can be easily reordered, added to, or streamlined. The result is a highly modular and configurable assembly approach to the controller layer, avoiding the development and maintenance overhead of unique controller servlets for each application.

In one embodiment, a site (or web application) is a collection of KSPs, each representing different areas and modules of the application, application logic, and various site-specific state diagrams that describe how the pages and application logic are integrated into the application flow. Pages and application logic can be thought of as modules that are assembled into a dynamic web application by the site-specific state diagrams. The Kinzan State Manager can manage this assembly and control.

3.3.3 Model Layer

In the Java Model 2 architecture, EJBs are the primary mechanism for managing the model layer. While extremely powerful and flexible, EJBs are also extremely complex to work with, especially for less technically sophisticated developers. This complexity is compounded by the need to tightly couple the controller servlet and JSP view layers to the EJB layer.

In one embodiment, the Kinzan Services Manager provides a layer of abstraction over many different types of services. Preferably, all sites have available to them various shared services (security, authentication, personalization, logging, etc.), business logic in the form of EJBs (or other component models), access to databases, and access to external applications and to external services such as e-mail, payment processing, etc.

Preferably, the Services Manager also provides a mechanism to dynamically look up services and bind to them, providing a flexible and accessible bridge between the model layer and the controller and view layers.

3.4 Module Reuse

In one embodiment described in this section, the collection of KSPs, widgets, wizards, and associated modules make up a site. Each site is actually part of a large hierarchy of sites. Resources associated with sites at upper levels are available for use to all lower levels. However, lower levels may provide their own variant on a resource that overrides the more global declaration.

Resource references may be resolved in a manner similar to that used with search paths, with lower levels having higher precedence. The result is the ability to build libraries of resources, streamlining development by encouraging reuse of common modules.

3.4.1 Platform Level

At the top of the hierarchy is the Platform Level. This level contains generic resources (e.g., widgets, style definitions, structure definitions, container groups, templates, etc.) that are visible to all applications and pages developed on the platform.

Changes or additions at this layer are by default automatically immediately applied to all applications and all KSPs within them. The exception is if a particular platform resource has been overridden at the site or KSP level.

3.4.2 Site Level

The Site Level contains all resources and sublevels that are associated with a particular site/application. At this level are template bundles that represent all the brandable resources in a particular application.

Resources at this level override identically named resources declared at the platform level. For example, if the platform provides a generic `<widget type="stockQuote">` widget that an application developer chooses to replace with a different implementation, the application developer may declare his own `<widget type="stockQuote">` widget at the site level. Although the application developer may also implement a different `<widget type="myStockQuote">` widget to avoid naming conflicts, that would require modification of all references to `<widget type="stockQuote">` within the various KSPs in the application.

For situations where an application is being developed from scratch, this is not an issue. However, the more normal case will be an application being struck from a generic reference implementation of the application, and custom configured for a particular client. In this case, overriding the resource at the application level applies the customization to the entire application at once.

Sites are modeled as collections of KSPs. All resources made available at the site level (widgets, templates, etc.) are visible to all KSPs in the site.

Sites may also be bound to parent sites. All resources available to parent sites are recursively available to the child site. This greatly facilitates the development of resource libraries to speed site development.

3.4.3 KSP Level

The KSP Level actually contains all the resources required to render a particular page, including (if necessary) other KSPs. The files required for the KSP rendering pipeline are described in Section 4. The process of developing and loading these files into the Rendering Service is described in the "Kinzan Server Pages Developer's Guide".

3.4.4 Widget Level

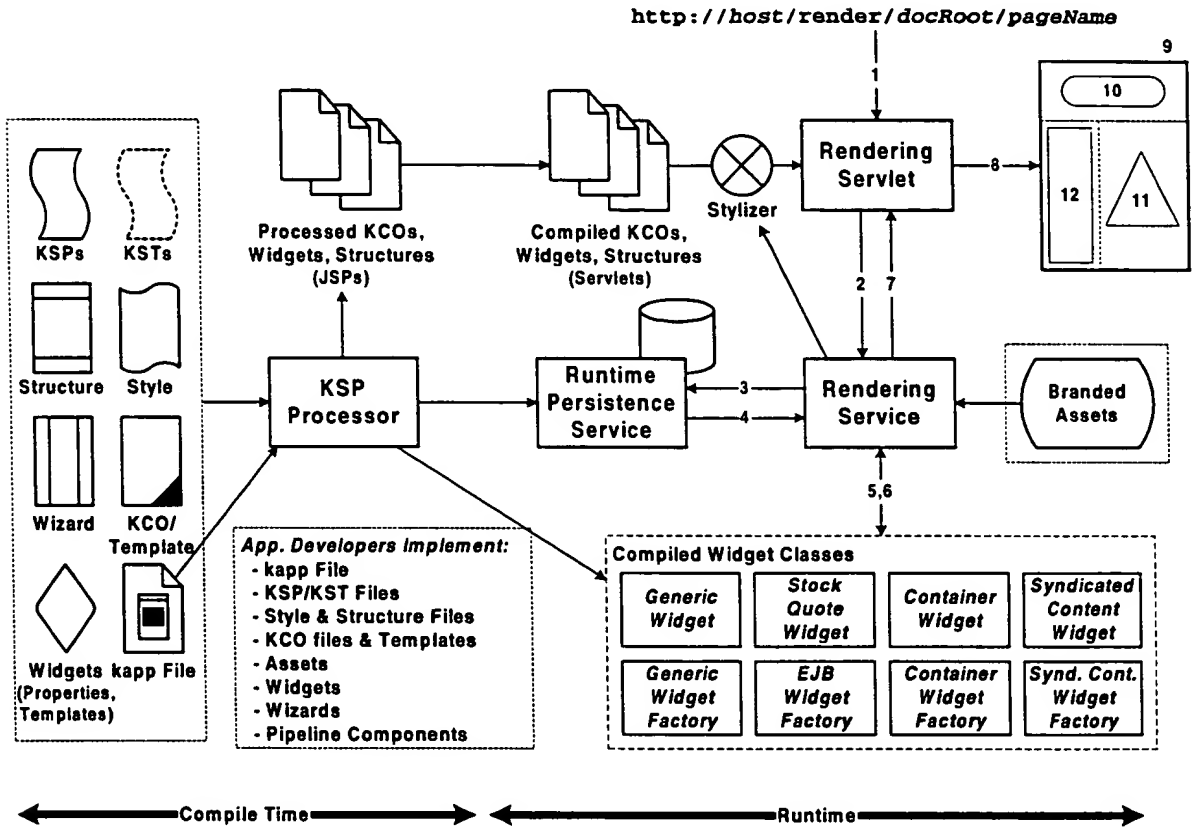
The Widget Level is the termination point for the KSP hierarchy. It may be contained at any level in the hierarchy (platform, application, or KSP levels), and may refer to other widgets.

Widgets are used to manage presentation and dynamic content (including localization), and to interface to the application logic layer. Description of the widget framework expected by the KSP Framework is described in the "Kinzan Widget Developer's Guide".

Please note that widgets may have their own resources and sublevels (including other widgets), but the widget is responsible for assembling and delivering the appropriate content and presentation from those resources.

4 KSP Processor Ov rvi w

The following figure gives a high-level overview of one embodiment of the Rendering Service, and will be referred to throughout this section and the next. This section describes the process of loading pages and modules into the runtime environment, while the next section describes how the various page modules are processed to dynamically generate the final page.



Sites are defined as a grouping of pages, as described by a Kinzan Application (kapp) file for the site. The kapp file is effectively the "makefile" for the site.

Each page is described by a KSP. There are three major file types which recursively define the presentation and content of each page. These are:

- KSP – Defines a page by declaring the structure, style, and contents to be used to assemble the page. Content is typically declared to be some combination of KCOs, KSPs, and/or widgets.
- KST – Defines a template page by declaring the structure, style, and default contents to be used when assembling pages based on the template.
- KCO – Defines presentation and content for a page or section (typically an unstyled JSP fragment). Also used as templates for widgets. A KCO may also include KSPs or other KCOs using widgets (Note: HTML is considered to be degenerate JSP, meaning KCOs may also contain unstyled HTML fragments)

Support files that are used in compiling the KSP are:

- Structure files – An XML file defining the structure and placement of content on a page or section.
- Style files – An XML file defining the style. Style is managed using a Cascading Style Sheets (CSS) – like syntax, and is applied by the Rendering Service on the server. This gives developers most of the

benefits of CSS, while maintaining compatibility with older browsers that do not have native support for CSS.

- Asset files – These are leaf objects that are included on a page via widgets. They include but are not limited to JSP, TXT, HTML, and image files. Assets may have variants that are dynamically resolved based on the current style, locale, and/or target device. An example may be graphics used for a navigation bar (`HomeButton`) that are different colors depending on the style being applied (`home_fire.jpg`, `home_ocean.jpg`, etc.). In this case, references to the `HomeButton` image asset will be dynamically resolved to the correct variant based on the style being used on a particular page.

The support files available to a site are defined in the kapp file for the site, which contains site-specific information. The kapp file is roughly the equivalent of the "makefile" for the web site. A site also inherits all the support files that are available to its parent sites.

At load time, the kapp file is loaded into the runtime environment, which in turn drives the recursive parsing, compiling, loading of the required structure files, style files, etc as compiled Java servlets. During this process, the compiler converts structure and KCO files into JSP fragments, which are then available to be stylized and recursively assembled by the runtime Rendering Service.

At compile time, the compiler parses a given KSP file and recursively loads the required sections into the runtime environment (as needed) for the Rendering Service to properly assemble the various modules that constitute the final page. If KCOs are included, those files are also parsed, compiled, and loaded. Other content is expected to have been already loaded in the runtime environment by the kapp file (i.e., assets, structures, and styles, etc.).

The end result of loading, processing, and compiling the various application modules is a runtime environment that is fully configured to run the application.

Please refer to the "KSP Developer's Guide" for examples of the files that are used to build a KSP, and the "Kinzan State Manager Developer's Guide" for examples of how pages are connected to application logic via state diagrams.

5 KSP Rendering Overview

The KSP Processor described in the previous section is responsible for processing and compiling site pages and site modules into the Rendering Service and State Manager. In one embodiment, the Rendering Service is responsible for dynamically assembling KSPs when requests are sent to the web server.

In this embodiment, as described in this section, the Rendering Service is fully dynamic, and may thus (with proper tools) be manipulated and configured on the fly, without having to recompile site pages or site modules. This rapid configurability may be a useful feature of the KSP Framework.

5.1 Inputs To Rendering Service

The KSP Processor (described in the previous section) is the primary means for loading various components into the runtime environment. Once there, components may be manipulated and configured. However, it is highly recommended that this only be done with tools that have been designed to do so in a safe way (the sample PageManager application that is included in the Fountainhead distribution is an example of an application that is used to manipulate the runtime of other applications).

5.2 Rendering Service

The Rendering Service internally uses the Model-View-Controller (MVC) paradigm to render pages.

The Model is the set of Java classes supplied with the widgets. These Java classes are responsible for creating the widgets, retrieving any data required by them and implementing any business logic. Note that internal to the rendering environment, all components (pages, widgets, containers, etc.) are modeled as widgets.

The View is the visual representation of the widget. In the rendering system, this is embodied in the corresponding widget template KCO file, which is processed into a JSP file for the widget. When the processed KCO file is compiled into a servlet and executed, the servlet interrogates the widget and uses the data contained in the widget to create a visual representation of it.

The Controller is the rendering servlet. The rendering servlet processes requests by matching a request to a model and view.

The figure above presents a detail depiction of how the widget Java classes and servlets are used by the rendering system to process a request. A brief description of each component follows:

Rendering Servlet: The rendering servlet processes requests to render pages. It forwards the request to the Rendering service and invokes the appropriate servlet for the page.

Rendering Service: The rendering service is responsible for building the model of a page. It uses the Runtime Persistence service to retrieve data from the Runtime datastore. For each component on the page, the rendering service uses this data to invoke the appropriate widget factory and create the corresponding widget. The page model is returned as a tree structure where the root of the tree is the page widget.

Runtime Persistence Service: The Runtime Persistence service encapsulates all operations for the Runtime datastore as a service. The datastore is only accessible through the Runtime Persistence Service.

Widget Factory: The widget factory is an implementation of the factory pattern. The Rendering Service uses the widget factory to instantiate a widget without knowing which type of widget is being created or what steps are required to create the widget. The widget factory is managed by the Rendering Service. The Rendering Service creates and initializes the widget factory once during the first request for a particular widget. The Rendering Service uses the same widget factory for any subsequent requests for the same widget type.

Widget: The widget object contains all the data required to render the widget. At runtime, a widget also provides access to all its defined properties.

Widget Proxy: The widget proxy is used by the Rendering service as a placeholder for the widget. The Rendering service uses information in the widget proxy to style and place the widget on the page. The Widget Proxy relieves the widget designer from maintaining page-specific information.

Widget Template KCO: The widget template KCO (once processed and compiled into a servlet) translates the data stored in a widget instance into its visual representation.

5.3 Building And Using Widgets

For details on how to build and use widgets, please refer to the "Kinzan Widget Developer's Guide".

6 Kinzan State Manager Overview

Managing web applications is separated into two broad areas: Application navigation and application flow.

6.1 Application Navigation

The first is application navigation, which reflects the hierarchical sitemap to the various application modules. For instance, if a user clicks on the icon for the home page, the user will be taken to the home

page for the site. Developers typically enable and manage this sort of free form navigation by using a Navigation widget for the site, which manifests itself as a Navigation Bar on the web page.

Navigation is appropriate for sections of a site that are not related in a common workflow. That is, sections of a site that may be navigated to directly, without requiring a specific sequence of events.

6.2 Application Flow

The second broad technique for managing web applications is combining application-specific functions within modules (e.g., KSPs) with components that represent application logic. In one embodiment of the KTP Framework, pages and application logic are linked together into an application flow with the Kinzan State Manager. Examples of applications of the State Manager are outlined in the "Kinzan State Manager Developer's Guide", a portion of which is presented here.

The State Manager (in one embodiment) is a very powerful and flexible tool for assembling web applications. Using the State Manager, developers can rapidly connect web pages (typically Kinzan Server Pages (KSPs)) and application logic to configure their application.

The State Manager supports flexible application configuration by cleanly separating state management from the presentation layer, and providing a mechanism to chain application logic modules to connect application pages. It also provides for management of state at the user and request level, avoiding many concurrency issues that plague less sophisticated state management systems (including duplicate request handling).

Application logic is implemented in modules using Java objects called pipeline components. These modules allow for "plugable" components that may be assembled dynamically based on user input and the current state. Components may be developed at a more atomic level and chained together, giving more flexibility in modifying or customizing applications for different users. The State Manager also provides for convenient and powerful field validation by these components.

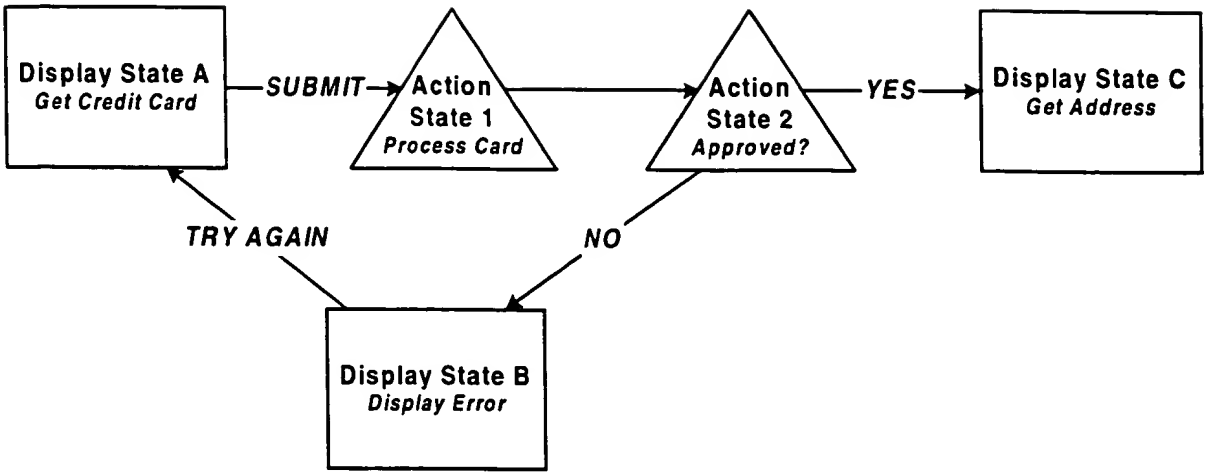
The State Manager provides basic transaction management across pipeline components, and facilitates linkages to business logic (usually Enterprise Java Beans (EJBs)) and the Kinzan Services Manager.

Applications may be defined to have many mini-applications (or wizards) within them, allowing better logical segmentation of application functions. This segmentation encourages reuse of wizards within and between applications, streamlining application development. The State Manager supports coordination and handoff between wizards.

Wizards are defined using XML, allowing for flexible configuration of wizards by less sophisticated developers, with easy integration into support tools.

6.2.1 What Is A State Diagram?

A state diagram (interchangeably referred to as a "wizard") represents discrete steps in an application flow, with decision points that trigger transitions to different sections of the application.



For example, in the figure above, the application begins in "Display State A". When the user enters their credit card number and generates a "SUBMIT" event by pressing submit on a form, the application moves to "Action State 1". "Action State 1" processes the credit card number and forwards the results to "Action State 2". "Action State 2" applies business rules to the results of the credit card processing to determine whether the merchant will accept the credit card.

If the rules in "Action State 2" generate a "YES" event, control is passed to "Display State C" which presents the user with a form to enter their address. If the rules in "Action State 2" generate a "NO" event, control is passed to "Display State B" which presents the user with a rejection message and gives them the opportunity to enter another credit card number.

In "Display State B", if the user chooses to enter another credit card number (perhaps by pressing a "Try Again" link), a "TRY AGAIN" event is generated and control is passed back to "Display State A" for the process to begin again.

6.2.2 Advantages Of Using A State Diagram

HTTP is a stateless protocol. Once a web server transfers a web page to a web browser, it has no knowledge about how pages are related to one another. This is one reason why links are embedded within web pages instead of being managed by the server.

By utilizing a state manager (with appropriate state diagrams), application developers can describe sophisticated relationships within an application, without being forced to manage all sorts of tricks involving manipulation of the actual web pages.

Once developers begin to model their applications using traditional state diagrams, they immediately benefit from enhanced documentation, manageability, flexibility, and reuse across applications.

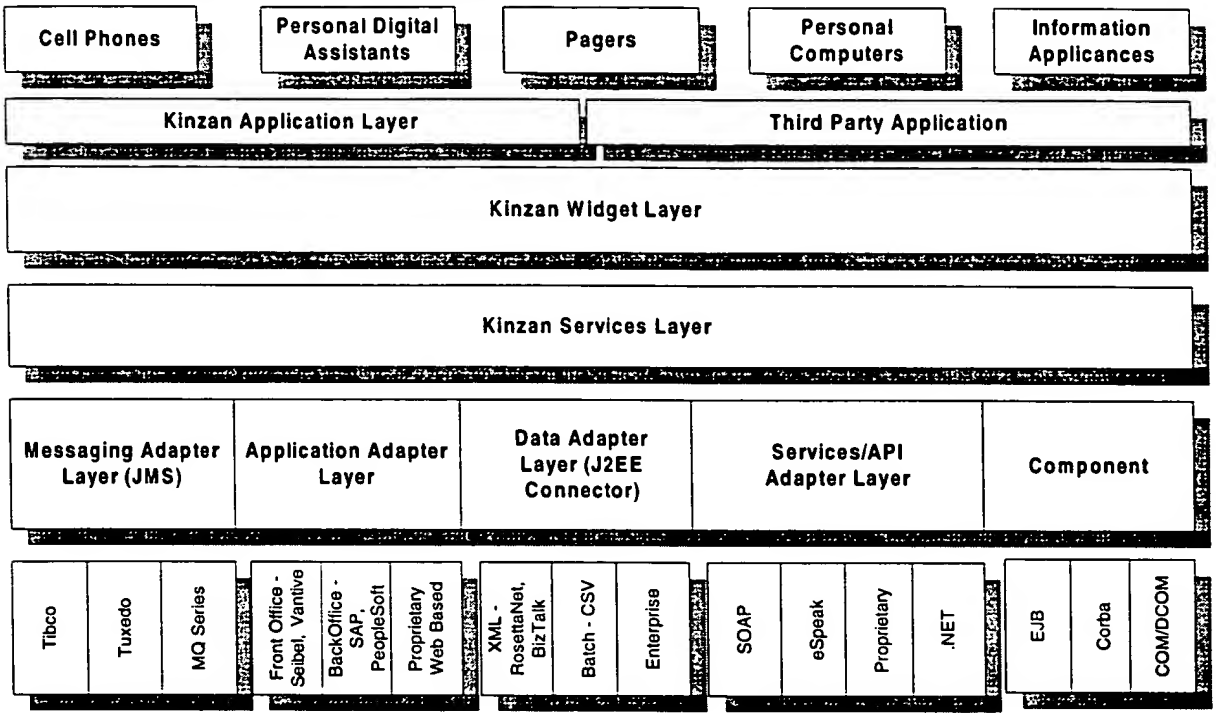
For instance, in the example in the previous section, a developer could easily replace "Action State 2" with a different piece of business logic that changes the criteria used to approve or deny a credit card transaction. They could do so by replacing a single component, rather than having to dig through a much larger piece of application logic that was distributed through lots of presentation code.

If, in the future, the application needs to be more sophisticated to support multiple approval components for different merchants, the modification to the application would be straightforward. The developer could insert a piece of application logic between "Action State 1" and "Action State 2" to determine which of many possible approval components should be invoked for a particular merchant and trigger that approval component, all without impacting the other components of the application (logic or presentation).

6.2.3 Building And Using A State Diagram

For details on how to build and use state diagrams, please refer to the "Kinzan State Manager Developer's Guide".

7 Kinzan Services Manager Overview



7.1 Dynamic Service Resolution And Binding

7.2 Service Adapter Layers

7.2.1 Data Integration
(JDBC, LDAP, etc.)

7.2.2 Component Integration
(EJB, CORBA, COM, etc.)

7.2.3 Application Integration
(ERP, CRM, legacy, etc.)

7.2.4 Messaging Integration
(JMS, eterprise messaging, etc.)

7.2.5 Internet Services Integration
(SOAP, XML, etc.)

8 Kinzan Shared Services Overview

8.1 Kinzan Security Framework

Token-based mechanism to extend Java security down to the method parameter level.

8.2 Kinzan Common Authentication Framework

Single sign on. Tightly coupled with security framework.

8.3 Kinzan Personalization Framework

Shared personalization facility.

8.4 Kinzan WebTop Framework

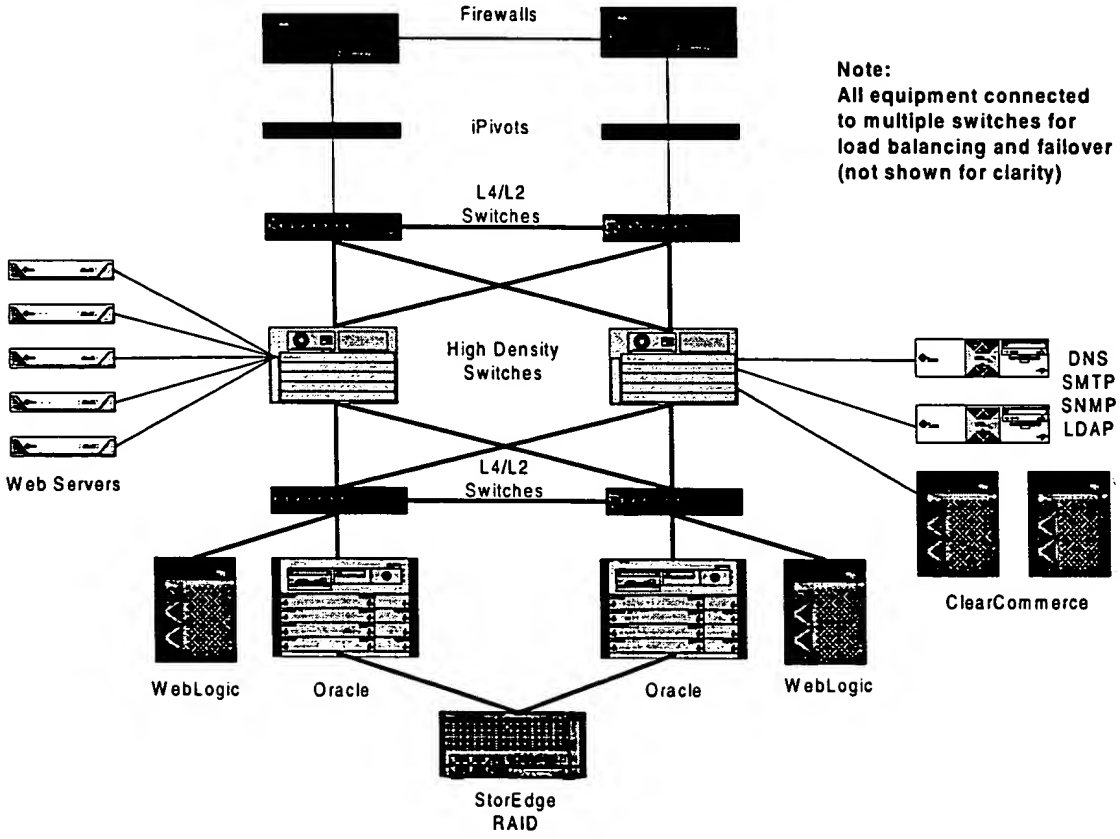
Shared and rapidly brandable user interface widgets

8.5 Kinzan Logging Framework

Shared application logging

9 Kinzan Infrastructure Platform Overview

9.1 Production Environment



9.2 Development Environment

9.3 Testing And Release Environment

10 VADIP Ov rvi w

Use of Kinzan Archives (KARs) to support deployment between development environments and shared hosting environments.

Tightly coupled with security framework

11 Elements Of Static And Active Web Applications

Embodiments of various components are discussed within this section, providing exemplary implementations thereof.

11.1 Style

Style elements are those that define the particular rendering of display elements, such as color, font, etc. Style markup is specific to the target platform (HTML, WML, etc.) The KTP Framework implements an extended variant of Cascading Style Sheets (CSS), although the stylizing is done on the server so as to not require CSS support on the client.

11.2 Structure (Layout)

Structure describes the high-level layout of the presentation, and is typically consistent from page to page in an application. An example would be a description that specified a header region across the top of a page, a navigation region down the left of a page, and a content region in the remaining of the page.

11.3 Presentation

Within each zone of a page layout, various display elements are collected into a presentation for that zone. Presentation is modeled separately because similar content may be presented in different ways in different contexts. For example, sales data may be presented in graphical form or in a spreadsheet form without changing the content.

11.4 State Diagram (Wizard)

State diagrams (or wizards) represent how different parts of an application flow into each other, including application logic for determining which part of an application comes next. An example would be a check out wizard on an e-commerce web site, where the application flows through different steps of the check out procedure with various pieces of application logic making decisions on the way (i.e., credit card accepted or rejected, etc.)

11.5 Services

Services represent different ways to get at information and to store information. These include databases, messaging services, higher-level application components like EJBs, etc. Typically, the main challenges in working with services are finding the service that provides the information you need, then interfacing with that service to retrieve the information.

11.6 Related Concepts

Each of these high-level elements is more or less impacted by some related concepts.

11.6.1 Containers

Containers are collections of content and presentation, typically deployed within a page layout. Containers may contain widgets (see next item), JSP fragments, other containers, links to external resources, or any of these in combination.

11.6.2 Application Components (a.k.a. "Widgets")

Widgets are represented as embedded tags that represent dynamic functionality, presentation, and content on a page (e.g., `<widget type="shoppingBasket">`, `<widget type="stockQuote">`, etc.)

By encapsulating application components in tag libraries, HTML programmers may rapidly assemble and link active components in a given container, without requiring extensive programming experience in Java or JSP.

11.6.3 Localization

The elements that make up a KSP collection may need the capability to support final JSPs for multiple languages. Localization decisions may be made at compile time, or may be made dynamically at run time, depending on application requirements.

11.6.4 Capability

The future Internet will be a composite of many "non-traditional" devices, such as PDAs, cell phone, pagers, telephones, appliances, etc. The KTP Framework and rendering process allows for graceful extension to different capability devices.

11.6.5 Templates

Templates are collections of common resources and content that apply on a site-wide basis. Templates are usually used to contain all "brandable" aspects of an offering, so that an application may be rapidly private-labeled for different partners, and customers may select from multiple branding motifs for their site.

Kinzan Server Pages Developer's Guide

1 Introduction

Programming with Kinzan Server Pages (KSPs), in one embodiment, involves a declarative approach, where predefined modules are assembled to build a given page.

The KSP environment, in one embodiment, requires the various modules that are necessary to compose a site and its pages be compiled and loaded into the runtime environment (see the "KTP Framework Overview" document for background on the rendering process).

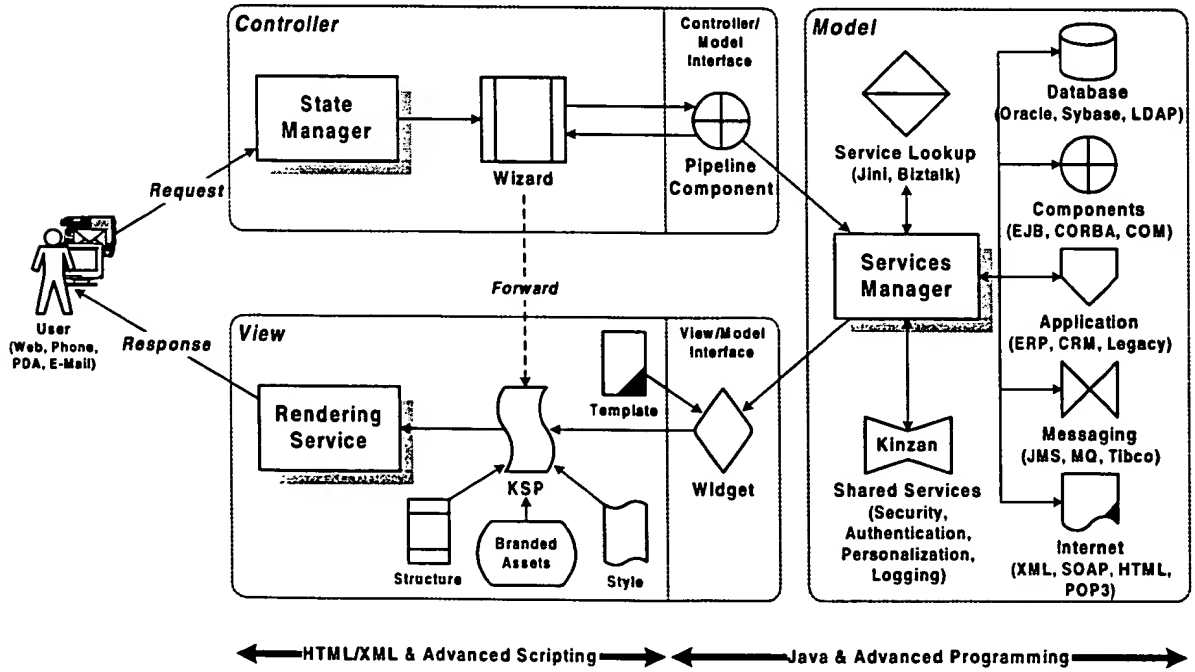
By separating the run-time and compile-time environments, the developer benefits from the ability to rapidly assemble precompiled modules when building a page, without the performance penalty normally associated with interpreted runtime environments.

In addition, the developer benefits from the ability to manage and reuse page elements across several pages and sites. By doing so, a site developer may centrally manage page content as well as site-wide look and feel without the need to edit each individual page.

Finally, the developer benefits by decoupling the design of a page design from the output device or locale. The page compiler and runtime are responsible for generating and assembling compiled modules that are appropriate to the target device or locale.

While populating and manipulating the runtime environment by hand is possible, this is akin to programming in machine language. In one embodiment, the loader/compiler provides a set of easy to manage files that describe site and page information in an easy to manipulate format, with object references and names being resolved at compile-time.

2 The Big Picture



In one embodiment, Kinzan Server Pages and Kinzan Widgets represent the "View" in Kinzan Technology Platform (KTP) architecture.

In contrast to the use of monolithic Java Server Pages (JSPs) in a typical Java Model 2 architecture, some embodiments of KSPs may allow the easy modularization and dynamic assembly of structure, style, dynamic presentation, and branded assets. Since pages are dynamically assembled and rendered by the Rendering Service, KSPs may allow for extremely easy and rapid deployment of pages to different target devices (wireless phones, PDAs, web browsers, etc.) with different styles and brand and across different locales.

In some embodiments of the KTP architecture, interfacing between the view and model layers (typically the most technically sophisticated and daunting component of a multi-tier architecture for non-programmers) is consolidated into easily reusable widgets (see the "Kinzan Widget Developer's Guide"). This allows graphics designers, information architects, and web developers to focus on creating compelling customer experiences, and not the nitty gritty of multi-tier application development.

In some embodiments, all the various modular artifacts in the KSP framework can be inherited and shared between different sites and applications, encouraging true reuse and consistency when assembling (versus building) applications.

The use of some embodiments of the Kinzan State Manager to apply a similar modular assembly approach to the controller tier is described in the "Kinzan State Manager Developer's Guide".

This document introduces the basics of the KSP development environment, including an overview of how applications are organized and loaded into the runtime environment for execution. More complex examples are provided in the developer's guides for other areas of the architecture.

3 Files and File Types

In some embodiments, as described in this section, there are two major files and various other support files required to build pages and sites.

The first required file is a Kinzan Application (kapp) file for the site. This file is an XML file that describes the site and assets that belong to the site (the equivalent of a "makefile"). The kapp file declares site information (such as the site name and document root), as well as assets that are available to be used in the various pages of a site (including structures, styles, widgets, and straight assets such as image files or HTML fragments).

The other major file type is the Kinzan Server Page (KSP) file, which describes a page by declaring the structure, style, and basic content for the page.

3.1 kapp File

The kapp file for a site is an XML file that provides site information and describes objects that are available to pages within the site. A more complete kapp file example is provided below. An introduction to the various sections of a kapp file follows.

3.1.1 Site Information

Each kapp file corresponds to one site, which in turn is assigned to a particular level in the runtime site/application space. The site information is supplied as follows:

```
1   <site name = "sitename" visibility="visibilityLevel">
2      <parent>parentSitename</parent>
3      <domainList>
4         <domain name="siteDomain.com">
5            <documentRoot>siteDocumentRoot</documentRoot>
6         </domain>
7      </domainList>
8      <parent>parentSiteName</parent>
9   </site>
```

1 Set the site's name. This must be unique. Visibility (public, private, or protected) establishes whether the site and its contents are visible to children sites.

3-7 Set the domain and document root for the site. (Used at runtime to access this site. i.e. `http://domain.com/documentRoot/...`). Note that a site may have many domains in the domain list.

8 Link the site to the parent site by name. (Optional)

At compile-time as well as runtime, the search path for the site/application includes the search root for the site/application each of its parent sites. The search root defines where files for the site may be found. The search order is as follows:

1. Check if the file exists at the root of the search root.
2. Check if the file exists at the `root/[subdir]` directory (see below).
3. Recursively check parent's site root, applying same rules

The type of the file being searched for determines the `[subdir]`. These are:

[subdir]	File Type
structure	Structures
style	Styles
ksp	KSPs
kco	KCOs
wizard	Wizards
asset	Other assets (images, JSPs, HTML)
resource	Misc. resources that do not require dynamically resolved variations based on style, target device, or locale.

For file types that support variants based on locale, style, and/or target device type, by convention variants are managed in subdirectories within the appropriate `[subdir]` with the pattern `[locale]/[style]/[target]`. This is a file management convention, and the path to variants (relative to `[subdir]`) must be explicitly called out when referring to them.

3.1.2 Structure and Style Mappings

Structures and styles are defined in subsequent sections. The kapp file determines which structure and style are available to be used for the site. Structures and styles names are mapped to their associated files and described as follows:

```
1   <structureList>
      .
      .
2     <structure name="structureName">
3       <variant target="HTML">structureFile.structure</variant>
```

```
4          <description>This is a structure</description>
5        </structure>
       .
       .
6    </structureList>

7    <styleList>
       .
       .
8        <style name="styleName">
9           <styleFile>styleFile.style</styleFile>
10          <description>This is a style.</description>
11       </style>
       .
       .
12   </styleList>
```

1-6 List of structures to make available to this site. Pages in the site may utilize structures defined at this level and structures defined at any parent level.

2-5 Define a structure. The name must be unique within this site. The file is searched for as specified above. The description is optional. A structure may have several variants based on a combination of target device type, style, and locale. The appropriate variant is dynamically determined when pages are rendered.

7-12 List of styles to make available to this site. Pages in the site may utilize styles defined at this level and styles defined at any parent level.

8-11 Define a style. The name must be unique within this site. The file is searched for as specified above. The description is optional.

At the time the structure file is loaded into the runtime environment, it is parsed into a container JSP file that reflects the same structure. In the runtime environment, it is actually represented as a widget instance that is sharable across this site and any descendent sites. It is this widget that drives the recursive assembly of a page.

Styles are parsed and loaded into the runtime environment. They are later applied by the Rendering Service to stylize all processed elements (KCOs, widgets, structures, etc.).

Structures and styles declared in the kapp file are simply loaded into the runtime environment and made available to this site and descendent sites. The KSP file for that page defines the style and structure to be used for a given page.

Structure and style files are discussed in detail below.

3.1.3 Asset Mapping

Assets are named resources that are available to be used within the site. Assets may have multiple variants based on the active style, locale, and/or target device type that a page is being rendered for. Assets are mapped to the appropriate file in the kapp file as follows:

```
1    <assetList>
       .
       .
2      <asset name="imageAssetname" visibility="visibilityLevel">
3         <variant target="HTML">image/assetfile.gif</variant>
4         <variant target="HTML" style="styleName">
5            image/styleName/assetfile.gif
6         </variant>
7         <variant target="WML">image/wml/assetfile.gif</variant>
8         <description>An asset</description>
```

```
9       </asset>
        .
        .
10  </assetList>
```

2 Specify the asset's name. This must be unique within the site.

3-7 Declare the different variants for the asset and the file they are bound to. Variants may be based on locale, style, and/or target device type.

8 Optionally, give a description of the asset.

Assets are allowed variations depending on style, locale, and target device combinations. Style is always preferred when finding the best match for an asset. A default (non style/locale/target-linked) variation is not required, but is recommended.

Grouping assets is a very powerful technique for supporting localization, branding, and personalization, since assets may be utilized generically when developing pages and applications, yet still deliver a customized presentation based on the person viewing a site.

3.1.4 Widget Mapping

Widgets are made available to a site by specifying information for their factory in the kapp file. In addition, widget instances declared in the kapp file are available to all pages in a site as a site widget. Site widgets may be referenced by any page in this site or children site, and be centrally managed by this site.

See the "Kinzan Widget Developer's Guide" for more information about using and building widgets.

Widget factories that are to be made available to a site are defined in the kapp file as follows:

```
1   <widgetFactoryList>
      .
      .
2       <widgetFactory name="widgetFactoryName">
3          <class>com.domain.WidgetFactoryClassName</class>
4          <widgetTemplate>
5             <variant target="HTML">widgetTemplate.kco</variant>
6          </widgetTemplate>
7          <property name="propertyName1" value="propertyValue1"/>
8          <property name="propertyName2" value="propertyValue2"/>
9          <widgetIncludeList>
             .
             .
10             <widget type="incWidgetFactoryName" binding="bindingName">
11                <property name="incPropertyName1" value="incPropertyValue1"/>
12                <property name="incPropertyName2" value="incPropertyValue2"/>
13             </widget>
                 .
                 .
14         </widgetIncludeList>
15      </widgetFactory>
      .
      .
16  </widgetFactoryList>
```

2 Define a widget factory. The name must be unique within this site.

3 Define the class that defines the interface for this widget.

4-6 Define the default template that should be used as the template for the widget presentation. Widgets templates may have many variants based on target device type, style, and/or locale, and may be overridden by widget instances.

7-8 Set default properties for the widget. A list of these parameters may be provided if needed. The `<property>` tag defines configuration values for the widget factory and are widget dependent, and may be overridden by widget instances.

9-14 Optionally, a widget may contain one or more other widgets, as defined in the `<widgetIncludeList>`. Included widgets are defined the same as other widgets, with the addition of a *bindingName* which is used by the parent widget to coordinate and communicate with the included widget. If included widgets are defined with the widget factory, they define the default include widgets for widget instances, and may be overridden at the instance level.

Shared site widget instances may also be defined in the kapp file. Uses for this may include defining a site-wide logo or copyright:

```
1   <widgetList>
      .
      .
2       <widget type="widgetFactoryName" name="widgetInstanceName">
3          <property name="instancePropertyName" value="instancePropertyValue"/>
4       </widget>
      .
      .
5</widgetList>
```

2 Define the widget instance by name and provides the factory the widget is based on.

3 Define the properties for the widget instance. For example, for an image widget this may be the image asset to display.

The syntax of defining a widget instance is equivalent across all files (KSP, KCO, etc.). The widget factory is referenced and widget template and properties are set. If a site-wide shared widget instance is being defined in the kapp file for the site, the instance is named. Other objects can then reference this shared widget instance via the *ref* widget type using *widgetInstanceName*.

3.2 Structure Files

Structure files capture the high-level layout of the page, with named references to the containers that are to be embedded in each zone. Collectively, these files represent the layout and content of the page.

The structure file is always searched for in the `structure` directory under the search root. At load time, for each structure defined in the kapp file the loader loads the appropriate data to describe the structure into the runtime environment. It also processes the XML describing the structure into a JSP file, which the Rendering Service utilizes to drive the runtime assembly of the page.

The structure file is an XML file containing tags similar to HTML, wireless markup language (WML), or any other markup language appropriate to the target device type. It may contain some leaf content (such as text and images) but generally only contains a basic page structure, including placeholders for content (called "zones") within that structure. These placeholders for content are specified using a `<zone>` tag.

The zone name is bound in the runtime environment at load time. Any object utilizing this structure needs to define the zones associated with its content by the same names as the zones in the structure. Note that style elements may be hard coded into the structure file.

A sample structure file and sample syntax may be found below.

3.3 Style Files

The style file is an XML format file that defines style elements to apply to pages that reference the style. The format of the style file closely follows that used in Cascading Style Sheets (CSS). A list of tags and the style attributes for each tag is specified, and multiple classes may be defined in a style. These classes allow for different variations of style attributes for given tags to be defined.

A useful application of this feature may be to redefine the font tag for different sizes and/or colors in a `HEADER` class or a `HIGHLIGHT` class. In this way, page designers can maintain the integrity of a style across an entire page.

The Rendering Service applies the style transformation to the flattened JSP file, resulting in a JSP that is ready to be compiled into a servlet for the final application. The style transformation may also include JSP fragments to allow runtime style configuration by the user.

All style transformation occurs on the server, allowing page designers to benefit from CSS-like capabilities when developing pages, without having to be concerned about browser compatibility issues.

A style file and an explanation of its syntax may be found below.

3.4 Building a Page

Sites are defined as a grouping of pages, as described by a kapp file for the site. Each page is described by a KSP. There are three major file types which recursively define the presentation and content of each page. These are:

- KSP – Defines a page by declaring the structure, style, and contents to be used to assemble the page. Content is typically declared to be some combination of KCOs, KSPs, and/or widgets.
- KST – Defines a template page by declaring the structure, style, and default contents to be used when assembling pages based on the template.
- KCO – Defines presentation and content for a page or section (typically an unstyled JSP fragment). Also used as templates for widgets. A KCO may also include KSPs or other KCOs using widgets (Note: HTML is considered to be degenerate JSP, meaning KCOs may also contain unstyled HTML fragments)

Support files that are used in compiling the KSP are:

- Structure files – An XML file defining the structure and placement of content on a page or section.
- Style files – An XML file defining the style. Style is managed using a Cascading Style Sheets (CSS) –like syntax, and is applied by the Rendering Service on the server. This gives developers most of the benefits of CSS, while maintaining compatibility with older browsers that do not have native support for CSS.
- Asset files – These are leaf objects that are included on a page via widgets. They include but are not limited to JSP, TXT, HTML, and image files. Assets may have variants that are dynamically resolved based on the current style, locale, and/or target device. An example may be graphics used for a navigation bar (`HomeButton`) that are different colors depending on the style being applied (`home_fire.jpg`, `home_ocean.jpg`, etc.). In this case, references to the `HomeButton` image asset will be dynamically resolved to the correct variant based on the style being used on a particular page.

The support files available to a site are defined in the kapp file for the site, which contains site-specific information. The kapp file is roughly the equivalent of the "makefile" for the web site. A site also inherits all the support files that are available to its parent sites.

At load time, the kapp file is loaded into the runtime environment, which in turn drives the recursive parsing, compiling, loading of the required structure files, style files, etc as compiled Java servlets. During this process, the compiler converts structure and KCO files into JSP fragments, which are then available to be stylized and recursively assembled by the runtime Rendering Service.

At compile time, the compiler parses a given KSP file and recursively loads the required sections into the runtime environment (as needed) for the Rendering Service to properly assemble the various modules that constitute the final page. If KCOs are included, those files are also parsed, compiled, and loaded. Other content is expected to have been already loaded in the runtime environment by the kapp file (i.e., assets, structures, and styles, etc.).

The end result of loading, processing, and compiling the various application modules is a runtime environment that is fully configured to run the application.

3.5 KSP Files

A KSP file declares a page's structure, style, and the contents for the zones described in the structure being used. A KSP file may also be used to describe a section in another KSP page. Top-level KSP pages are found in the `ksp` directory under the search root, while KSPs used to describe sections are found in the `asset` directory under the search root.

KSP files may also describe a page template (KST files). When loaded as a template, pages may be created from them at runtime that maintain widget references, while creating new widget instances for template elements that are defined as such. In this way, KSTs may be used to load a preformatted page with predefined elements, some of which are controlled at the parent level others of which are controlled at the site level.

KSP files specify the contents to be used when assembling a page. This content may be widgets, widget references, and/or container objects (KCOs).

3.5.1 Widgets

```
<widget type="widgetFactoryName" [style="optionalStyle"]>
   <widgetTemplate>[1]
      <variant>...</variant>
   </widgetTemplate>
   <property .../>
   <widgetIncludeList>
      <widget binding="widgetBindingName">
         .
         .
         .
      </widget>
   </widgetIncludeList>
</widget>
```

This tag defines a widget instance directly. Widget-dependant properties and templates may be supplied when instantiating a widget, otherwise they are inherited from the widget factory. If specifying an optional style, the widget instance representation will utilize it.

If the page designer instantiates a widget in a page, the page owns a local copy of the widget.

Note that a name for the widget instance is not required, and not recommended. The compiler automatically assigns a unique name to ensure that there are no collisions in the runtime environment.

Additionally, a `widgetIncludeList` may be specified for the widget instance if necessary.

3.5.2 Widget References

```
<widget type="ref" name="siteWidgetInstanceName" [style="optionalStyle"]/>
```

Shared widgets that are instantiated and named in the kapp file may be referenced by name using a widget of type `ref`.

The difference between a widget instance and a widget reference is that the page does not own a local copy of the widget. Rather, the widget is merely referenced, and it is instantiated elsewhere. Any changes to the widget by the widget owner will be reflected in all pages that reference this widget, unless overridden at this level While overriding properties and templates is supported, the developer should consider instantiating a new widget rather than providing a reference if such behavior is necessary.

3.5.3 Container objects

An example of a container object is a KCO:

```
<widget type="kco">
   <widgetTemplate>
      <variant target="HTML">kcoFile.kco</variant>
   </widgetTemplate>
</widget>
```

The KCO widget allows a page designer an easy way to include a template file that is not necessarily bound to a widget type. A KCO file allows someone to quickly design a section containing HTML, JSP, and other widgets or widget references, including other KCO's. KCO files are styled at compile time to utilize the style of the object they are contained in. They are searched for in the KCO directory under the search root at compile-time. A sample KCO file and its syntax is presented in the Appendix A.

A section KSP, on the other hand, is a more structured container that takes advantage of reusable components such as style and structure and may contain other KSPs, KCOs, and leaf widgets.

If flexibility is required and the section's content needs to be dynamic, including a KSP as a section is recommended. Otherwise, a simple KCO is acceptable, especially if no specific style is required and most of the content is static anyway.

Following is an example of referencing a section KSP:

```
<widget type="ksp">
   <property name="KSP" value="nameOfKSP.ksp"/>
</widget>
```

3.6 Wizard Files

What KSPs are to the view tier, wizards are to the controller tier. Wizards are used to assemble, organize, and connect the various modules which make up the controller tier. They are XML files that define the different states in an application, and how different events drive different application functionality.

Wizards specify both display states and action states. Action states use modular pipeline components to execute different elements of application logic, with the result of one operation driving the next operation. For instance, an `xApproveCreditCard` action state may invoke an `xShipProduct` action state if a transaction is approved, or invoke an `xRejectOrder` action state if the credit card is declined.

Once all backend processing is done, the wizard specifies which KSP is to be invoked in order to show the user the next step in the application. For example, in the previous example, an `OrderConfirmation` display state if the card is approved and the product shipped, and a `ReenterCreditCard` display state if the card is rejected.

Overall application flow is controlled via the wizard file, with pipeline components and KSPs available to be shared within and between applications to encourage reuse.

Wizard files and the Kinzan State Manager are described more fully in the "Kinzan State Manager Developer's Guide".

The wizards that are needed for an application are defined in the kapp file as follows:

```
1   <stateManager>
2      <defaultState wizard="defaultWizardName" state="defaultStartState"/>
     .
     .
3      <wizard name="wizardName">
4         <file>wizardName.wizard</file>
5      </wizard>
     .
     .
6   </stateManager>
```

2 Set the default starting wizard and start state for the application.

3-5 Defines the wizard name and file to be made available to the application.

Wizards defined in the kapp file are loaded into the runtime environment and are available to be invoked by KSPs. Like other modules, wizards are also inherited from parent sites to child sites, encouraging reuse.

3.7 Resource Files

Much of the power and flexibility of the KSP framework derives from the ability to various modules display appropriately to multiple target device types, multiple styles, and in multiple languages. This is possible because the Rendering Service can dynamically resolve the appropriate variant every time a KSP is displayed.

However, it is often the case that this sort of flexibility is not needed for all assets that an application needs.

For these situations, all files and folders in the top-level resource directory will be automatically deployed to the document root for the application so they may be directly referenced from within the various application components.

4 Docum nt Typ Definitions

The following examples further illustrate various embodiments in an illustrative manner.

4.1 Document Type Definition (DTD) For kapp Files

The DTD for kapp files in one embodiment is as follows:

```
<?xml encoding="US-ASCII"?>

<!ELEMENT kapp (site,(styleList?,structureList?,assetList?,widgetFactoryList?,
                widgetList?,KSTVisibilityList?,KSPVisibilityList?,
                stateManager?)*)>

<!ELEMENT site (domainList+,parent?)>
<!ATTLIST site
          name CDATA #REQUIRED
          visibility CDATA "public">

<!ELEMENT domainList (domain+)>

<!ELEMENT domain (documentRoot)>
<!ATTLIST domain
          name CDATA #REQUIRED>

<!ELEMENT documentRoot (#PCDATA)>

<!--
 Optionally Specify a parent site.  If not specified, the default 'root' site is
    used.
-->
<!ELEMENT parent (#PCDATA)>

<!ELEMENT widgetFactoryList (widgetFactory+)>

<!ELEMENT widgetFactory
    (class,description?,widgetTemplate?,property*,widgetIncludeList?)>
<!ATTLIST widgetFactory
          name CDATA #REQUIRED
          visibility CDATA "public"
          extendsFactory CDATA #IMPLIED>

<!ELEMENT class (#PCDATA)>

<!ELEMENT structureList (structure+)>

<!ELEMENT structure (variant+,description?)>
<!ATTLIST structure
          name CDATA #REQUIRED
          visibility CDATA "public">


<!ELEMENT styleList (style+)>

<!ELEMENT style (styleFile,description?)>
<!ATTLIST style
          name CDATA #REQUIRED
          visibility CDATA "public">

<!ELEMENT styleFile (#PCDATA)>


<!ELEMENT assetList (asset+)>

<!ELEMENT asset (description?,variant+)>
<!ATTLIST asset
          name CDATA #REQUIRED
```

```
          beanName CDATA #IMPLIED
          visibility CDATA "public">


<!ELEMENT widgetList (widget+)>

<!ELEMENT widget (description?,template?,property*,widgetIncludeList?)>
<!ATTLIST widget
          type CDATA #IMPLIED
          name CDATA #IMPLIED
          visibility CDATA "public"
          binding CDATA #IMPLIED>

<!ELEMENT widgetIncludeList (widget+)>

<!--
 Listing of KST files with explicit visibility.  If not listed here, the KST
    defaults to "public" visibility
-->
<!ELEMENT KSTList (KST+)>

<!ELEMENT KST EMPTY>
<!ATTLIST KST
          name CDATA #REQUIRED
          visibility CDATA "public"
>

<!--
 Listing of KSP files with explicit visibility.  If not listed here, the KSP
    defaults to "public" visibility
-->
<!ELEMENT KSPList (KSP+)>

<!ELEMENT KSP EMPTY>
<!ATTLIST KSP
          name CDATA #REQUIRED
          visibility CDATA "public">


<!--
 List of wizards available to this site.
-->
<!ELEMENT stateManager (defaultState?,wizard+)>

<!ELEMENT defaultState EMPTY>
<!ATTLIST defaultState
          wizard CDATA #REQUIRED
          state CDATA #REQUIRED>

<!ELEMENT wizard (file,description?)>
<!ATTLIST wizard
          name CDATA #REQUIRED
          visibility CDATA "public">

<!ELEMENT file (#PCDATA)>


<!ELEMENT description (#PCDATA)>

<!ELEMENT property EMPTY>
<!ATTLIST property
          asset CDATA #IMPLIED
          name CDATA #REQUIRED
```

```
          value CDATA #IMPLIED
          visibility CDATA "public">

<!ELEMENT widgetTemplate (variant*)>
<!ATTLIST widgetTemplate
          asset CDATA #IMPLIED>

<!ELEMENT variant (#PCDATA)>
<!ATTLIST variant
          style CDATA #IMPLIED
          target CDATA #IMPLIED
          locale CDATA #IMPLIED>
```

4.2 Document Type Definition (DTD) For KSP Files

The DTD for ksp files in one embodiment is as follows:

```
<?xml encoding="US-ASCII"?>

<!ELEMENT ksp (title,structure,style?,zoneList)>

<!ELEMENT title (#PCDATA)>

<!ELEMENT structure (#PCDATA)>

<!ELEMENT style (#PCDATA)>

<!ELEMENT zoneList (zone+)>

<!ELEMENT zone (widget)>
<!ATTLIST zone
          name CDATA #IMPLIED
>

<!ELEMENT widget (property*,widgetTemplate?,widgetIncludeList?)*>
<!ATTLIST widget
          type CDATA #REQUIRED
          name CDATA #IMPLIED
          binding CDATA #IMPLIED
          style CDATA #IMPLIED
>

<!ELEMENT property EMPTY>
<!ATTLIST property
          name CDATA #REQUIRED
          asset CDATA #IMPLIED
          value CDATA #IMPLIED
>

<!ELEMENT widgetTemplate (variant*)>
<!ATTLIST widgetTemplate
          asset CDATA #IMPLIED
>

<!ELEMENT variant (#PCDATA)>
<!ATTLIST variant
          target CDATA #IMPLIED
          style CDATA #IMPLIED
>

<!ELEMENT widgetIncludeList (widget+)>
```

4.3 Example kapp File

```
<?xml version='1.0'?>

<!DOCTYPE kapp SYSTEM "http://www.kinzan.net/dtd/kapp.dtd">

<!--
    Sample kapp file for "SampleSiteName"
-->

<kapp>

   <!--
      Site Information.
      If the site is "final", then it can't have descendants.
      A site is "public" by default, but it can be explicitly specified.
   -->
   <site name="SampleSiteName" visibility="public">

      <!--
         A site can belong to multiple domains, each specified by domain name
         and document root.
      -->
      <domainList>
         <domain name="www.sampledomain.com">
            <documentRoot>SampleDocumentRoot</documentRoot>
         </domain>
         <domain name="sampledomain.com">
            <documentRoot>SampleDocumentRoot</documentRoot>
         </domain>
      </domainList>

      <!--
         Optionally specify a parent site.  If none is specified, it will
         automatically inherit from the predefined base site "root".
      -->
      <parent>SampleParentSiteName</parent>

   </site>

   <!--
      List styles available to this site and descendants
   -->
   <styleList>

      <!--
         Each style has a required name and optional visibility.

         The visibility determines whether this widgetFactory is available to
         descendant sites and overridable (public), available but not overridable
         (final), or not available (private).
         If not specified, the default is public.

         The styleFile specifies the name of the XML style file that describes this
         style. It needs to be located in [searchRoot]/style/.

         The description is optional.
      -->
      <style name="SampleStyle" visibility="public">
         <styleFile>sampleStyleFile.style</styleFile>
```

```
      <description>This is a sample style</description>
   </style>
</styleList>


<!--
   List widget factories available to this site and descendants.
-->
<widgetFactoryList>

   <!--
      The widgetFactory has a required name attribute and optional visibility.

      The class specifies the widget factory class.

      The template consists of a list of variants based on target, style, and/or
      locale. (Currently, locale is not used).
      Target is required.  Style is optional.  Templates
      may be overridden at the widget instance level

      Widget templates are typically KCOs, which are referenced relative to the
      [searchRoot]/kco/. directory

      Parameters are optional and can also have visibility:
         public - parameter can be overriden by widget.
         final - parameter can not be overriden by widget.
         private - parameter is hidden from widget.  (Edit only)

      Properties defined with widget factories establish the default properties
      for the associated widget instances

      The widgetIncludeList lists any widgets that you would like to associate
      with this widget.  They need to be bound in the widget's template file.
   -->
   <widgetFactory name="SampleWidgetFactory" visibility="public" >
      <class>com.kinzan.app.sample.SampleWidgetFactory</class>
      <widgetTemplate>
         <variant target="HTML">sampleTemplateA.kco</variant>
         <variant target="HTML"
                  style="SampleStyle">sampleTemplateB.kco</variant>
         <variant target="WML">wml/sampleTemplateC.kco</variant>
      </widgetTemplate>
      <property name="nameA" value="valueA"/>
      <property name="nameB" asset="SampleAsset"/>

      <!-- You can optionally redefine the widget class definition here -->
      <property name="WIDGET_CLASS" value="com.yourpackage.YourWidgetClass"/>
   </widgetFactory>

   <widgetFactory name="panel">
      <class>net.kinzan.rendering.component.GenericWidgetFactory</class>
      <description>This is a sample panel widget</description>
      <widgetTemplate>
         <variant target="HTML">SamplePanel.kco</variant>
      </widgetTemplate>

      <!-- The widgetIncludeList is recursive, so any widget defined in here can
           also have its own widgetIncludeList.  The binding name needs to be
           bound in the widget's template file.   -->
      <widgetIncludeList>
         <widget type="text" binding="content">
            <property name="TEXT" value="This is the basic content"/>
            <property name="TITLE" value="Basic Title"/>
```

```
            </widget>
         </widgetIncludeList>

      </widgetFactory>

   </widgetFactoryList>


   <!--
      List structures available to this site and descendants.
   -->
   <structureList>

      <!--
         Each structure requires a name and has an optional visibility.

         Each variant listed can be based only on target (required) and/or locale.
         (Locale is currently not utilized).  The file needs to be located relative
         to [searchRoot]/structure/.

         The description is optional.
      -->
      <structure name="SampleStructure">
         <variant target="HTML">SampleStructureFile.structure</variant>
         <variant target="WML">wml/SampleStructureFile.structure</variant>
         <description>This is a sample structure.</description>
      </structure>

   </structureList>


   <!--
      List assets available to this site and descendants.
   -->
   <assetList>

      <!--
         Each asset is defined by a required name and optional visibility.

         A list of variants per asset specify variants based on style, target, and
         locale. (Locale is currently not utilized) Target is required.

         By convention, assets are categorized and placed into the following
         directory under the [searchRoot]/asset/. directory:

            image - Contains image assets (gif, jpg, wmp, bmp, etc)
            jsp   - Contains plain jsp fragments.
            file  - Contains other miscelanneous formats.

         The description is optional.
      -->
      <asset name="sampleAsset" visibility="final">
         <variant target="HTML">image/sampleAssetFile.gif</variant>
         <variant target="HTML"
                  style="SampleStyle">image/sampleAssetFileA.gif</variant>
         <variant target="WML">image/wml/sampleAssetFile.jpg</variant>
         <description>Sample Asset Description.</description>
      </asset>

   </assetList>

   <!--
      Lists the widgets available to this site and descendants.
```

```
-->
<widgetList>

   <!--
      Each widget has a required name and type, and optional visibility.

      The type defines the widget factory the widget instance is based on.

      Widget instances defined in the kapp file may be referenced elsewhere in
      the site by <widget type="ref" name="SampleSharedWidgetName"/>

      Each widget has a list of parameters with a required name and either a
      value or asset pointer.
      The asset must either be defined above or at a parent site and be public.
      Widget properties do not have visibility constraints.  (They are always
      public).
   -->
   <widget type="SampleWidgetFactory"
           name="sampleSharedWidgetName" visibility="public">
      <property name="nameA" value="valueA"/>
      <property name="nameB" asset="SampleAsset"/>
   </widget>

   <widget type="kco" name="sampleKCOSharedWidget">
      <description>This is a sample sitewide kco instance</description>
      <widgetTemplate>
         <variant target="HTML">sample.kco</variant>
      </widgetTemplate>
   </widget>

   <widget type="panel" name="sampleSitePanel">
      <description>This is a sample sitewide panel</description>

      <!-- You can optionally override widgets in the widgetIncludeList -->
      <widgetIncludeList>
         <widget type="text" binding="content">
            <property name="TEXT" value="This is the sitewide panel!"/>
            <property name="TITLE" value="Sitewide Title"/>
         </widget>
      </widgetIncludeList>
   </widget>

</widgetList>


<!--
   KSTList configures the list of KST templates' available to this site and
   descendents.

   The file needs to be located relative to [searchRoot]/kst/.
   It is currently not used.
-->
<KSTList>
   <KST name="SampleKST" visibility="final"/>
</KSTList>

<!--
   KSPList configures the list of KSP pages' available to this site and
   descendents.

   The file needs to be located relative to [searchRoot]/ksp/.

   It is currently not used.
```

```
-->
<KSPList>
   <KSP name="SampleKSP" visibility="private"/>
</KSPList>

<!--
   stateManager configures the list of wizards available to this site and
   descendents.

   The file needs to be located relative to [searchRoot]/wizard/.

   It is currently not used.
-->
<stateManager>
   <defaultState wizard="login" state="welcome"/>
   <wizard name="SampleWizard" visibility="public">
       <file>sample.wizard</file>
       <description>This is a sample wizard</description>
   </wizard>
</stateManager>


<!--
   By default files and directories in[searchRoot]/resource/. are automatically
   deployed relative to the document root.

   This is useful for files (images, etc.) which do not need to be managed as
   assets.
-->
```

4.4 Sample Structure File

```
1   <?xml version="1.0"?>
2   <html>
3      <head>
4         <title><PageTitle/></title>
5      </head>
6      <body>
7         <table width="100%" border="1">
8            <tr>
9               <td>
10                 Order: A,B,C
11              </td>
12              <td>
13                 <zone name="A"/>
14              </td>
15           </tr>
16           <tr>
17              <td>
18                 <zone name="B"/>
19              </td>
20           </tr>
21        </table>
22        <table class="WINTER">
23           <tr>
24              <td>
25                 <zone name="C"/>
26              </td>
27           </tr>
28        </table>
29     </body>
```

```
30  </html>
```

The structure file is an XML file defining a page structure. Placeholders called zones (Lines 13, 18, and 25) are defined that are bound to content in a KSP file. Some notes on the syntax of the file follows.

4 Note the special `<PageTitle/>` tag. This is replaced with the page's title (as defined with a `<title>` tag in the KSP file) at runtime.

10 This structure defines some static text that will be included into all pages that use this structure.

13 Zone tag. (Also 18, 25)

22 The `CLASS` attribute specifies that this section should use the `WINTER` variant of the style in effect. If such a variant does not exist, the default variant will continue to be in effect, unless content in the zone overrides it.

4.5 Example Style File

```
1   <?xml version="1.0"?>
2   <Style name="Style1" >
3      <class>
4         <table>
5            <attribute name="bgcolor"  value="#AA00AA"/>
6            <attribute name="bordercolor" value="#FF0000"/>
7         </table>
8         <td>
9            <attribute name="bgcolor" value="white"/>
10        </td>
11        <body>
12           <attribute name="bgcolor" value="red"/>
13        </body>
14        <font>
15           <attribute name="face" value="helvetica"/>
16           <attribute name="size" value="4"/>
17           <attribute name="color" value="red"/>
18        </font>
19     </class>
20     <class name="winter">
21        <table>
22           <attribute name="bgcolor"  value="#AA0000"/>
23           <attribute name="bordercolor" value="#0000FF"/>
24        </table>
25        <td>
26           <attribute name="bgcolor" value="white"/>
27        </td>
28        <body>
29           <attribute name="bgcolor" value="blue"/>
30        </body>
31        <font>
32           <attribute name="size" value="2"/>
33           <attribute name="color" value="blue"/>
34        </font>
35     </class>
36  </style>
```

2 Defines the style and style name.

3-19 Defines the default style elements.

20-35 Defines the `winter` variant of this style.

The style file is an XML format file that defines various style elements associated with html tags. Any valid html tag and style attribute is valid. These are loaded into the runtime environment to be applied at runtime. Any tags without style attributes in the current class will be expanded from the default class. If it doesn't exist in the default class, it will be expanded from the style in effect in the outer scope. This is detailed in the runtime environment documentation.

4.6 Example KSP File

```
1   <?xml version="1.0"?>
2   <!DOCTYPE ksp SYSTEM "http://www.kinzan.net/dtd/ksp.dtd" >
3
4   <ksp>
5      <title>Sample Page Title</title>
6      <structure>
7         Structure1
8      </structure>
9      <style>
10        Style1
11     </style>
12     <zoneList>
13        <zone name="A">
14           <widget type="image">
15              <property name="IMAGE" asset="rose"/>
16           </widget>
17        </zone>

18        <zone name="B">
19           <widget type="kco">
20              <property name="file" value="sample.kco"/>
21           </widget>
22        </zone>

23        <zone name="C">
24           <widget type="ref" name="copyright"/>
25        </zone>
26     </zoneList>
27  </ksp>
```

5 Declares the title for the page. (This is substituted into the `<PageTitle/>` tag of the structure file at runtime.

6-8 Declares the main structure for the page.

9-11 Declares the main style for the page.

12-26 Declares the content to place into the zones in the structure. Note that the zone names must match the zone names in the structure, otherwise the content is not displayed. Content may be one of several types, as discussed in Section 2.5 of this document.

4.7 Example KCO file

```
1   <table class="SUMMER">
2      <tr><td>
3         <widget type="image">
4            <property name="IMAGE" asset="rose"/>
5         </widget>
6      </td></tr>
7      <tr><td>
```

```
8          This is a KCO with an image widget!
9        </td></tr>
10   </table>

11   <% out.print( "Just some JSP code to throw it in!" ); %>
```

3-5 Defines and includes an image widget instance, containing the image specified by the asset `rose`.

11 Some straight JSP code.

KCO files are basically JSP files that may contain widgets and that are styled. Note that the `<widget>` tag is similar to the one included for the kapp and KSP files.

KCO files are compiled into JSPs, where the widget definitions and references are replaced by code to be executed at runtime to properly include the widget.

5 Exemplary Components and Activities

The following descriptions illustrate various embodiments of various components and activites in an illustrative manner.

5.1 Loader

Load time consists of parsing the kapp file, copying referenced assets to the deployment directory, generating appropriate structure JSP files, processing KCO files, and configuring the runtime environment with site, style, structure, and widget information.

To run the loader, you will need to ensure that you have a properly formatted kapp files and all files referenced in the kapp file are in their appropriate locations.

Run the loader by specifying the a property file, the kapp file to load, and an optional overwrite mode:

```
java net.kinzan.rendering.tool.kspc.LoadKApp loader.properties kappfile.kapp
[overwrite]
```

If run in overwrite mode, the loader will reprocess and overwrite all files without regard for modification time. The default mode is to only overwrite files if they have been modified since the last load.

When reloading a site's kapp file elements may not be removed, only modified. Consequently, it is safe to reload a kapp file using the loader. Any elements that the loader finds in the kapp file that matches an existing element will be updated, and new elements added.

The loader requires the following in the classpath:

`xerces.jar`
`xalan.jar`
`antler-2.7.0`
`TOPLink.jar`
`TOPLinkX.jar`
`Tools.jar` (Toplink package)
`jgl2.1.0.jar`
`816classes12b.zip` (Oracle JDBC driver)

5.2 Compiler

After running the loader, you are ready to compile KSP pages. Ensure that any files your KSP references exist in the correct location, and then run the compiler on the KSP file:

```
java net.kinzan.rendering.tool.kspc.KSPCompilerApp loader.properties sitename
kspfile.ksp [overwrite]
```

This process will create JSP files in the asset directory of the deployment root specified in the properties file. If replacing an existing page, the "`overwrite`" option is required. The classpath and requirements are equivalent to those for the loader.

You may also choose to compile all KSP files in your ksp directory at once. The `GenSiteApp` application will traverse the directory structure inside a site's ksp directory and compile all the pages it finds into the directory structure they are in:

```
java net.kinzan.rendering.toolkspc.GenSiteApp loader.properties sitename
[overwrite]
```

5.3 Properties and Notes

The properties file for the loader applications require several elements to be defined. These are the deployment root and the search root for each site:

```
# Deployment root
deployment.root=E:/netscape/server4/docs
sample.search.root=E:/KTP/sample
```

Only one deployment root can be defined. This is the root location to where the loader/compiler applications should copy processed assets and resources. Files will be placed into the deployment root directory under a directory named by the site name. Individual page-specific elements will be placed into page-named directory structures under each main directory. For example, a KCO, sample.kco in a page named "ExamplePage" in site "sample" would be processed and placed in:

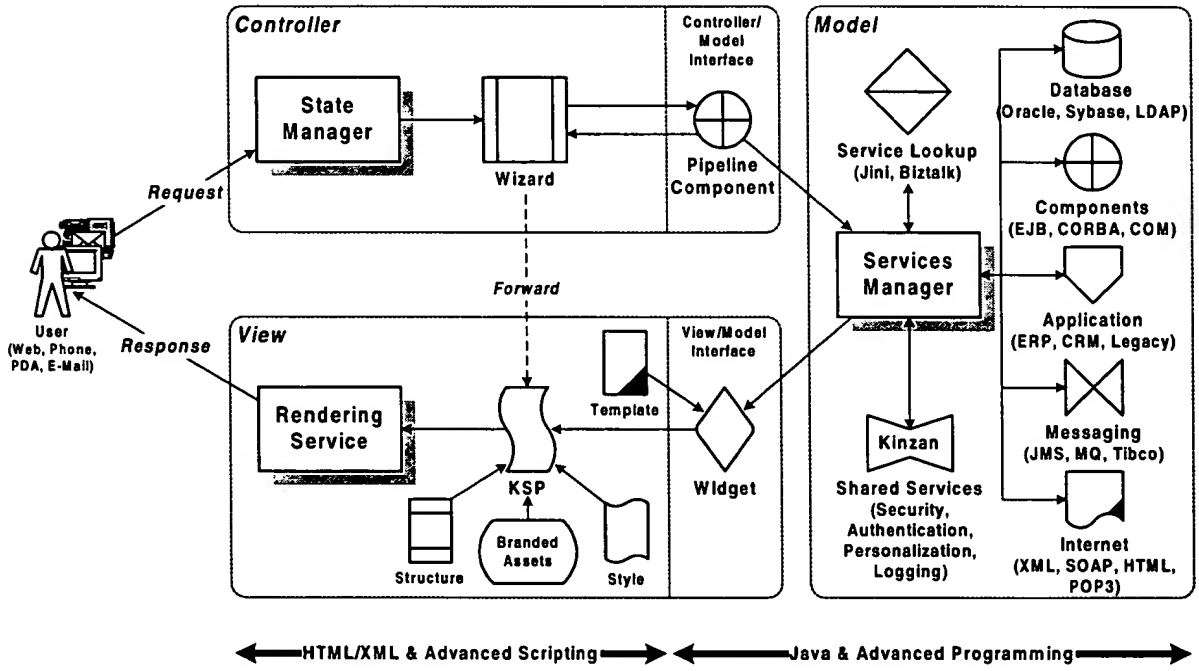
`[`*`deployment.root`*`]/sample/kco/ExamplePage/`*`sample.jsp`*.

Kinzan Widget Developer's Guide

1 Introduction

This document provides an overall view of how widgets are used by the runtime Rendering Service in one embodiment. It also describes an API that may be used to create and customize widgets and gives several examples (including example code).

2 The Big Picture



As discussed in the "KTP Framework Overview" document, Kinzan Server Pages and Kinzan Widgets represent the "View" in Kinzan Technology Platform (KTP) architecture in one embodiment.

In contrast to the use of monolithic Java Server Pages (JSPs) in a typical Java Model 2 architecture, KSPs allow the easy modularization and dynamic assembly of structure, style, dynamic presentation, and branded assets. Since pages are dynamically assembled and rendered by the Rendering Service in one embodiment, KSPs allow for extremely easy and rapid deployment of pages to different target devices (wireless phones, PDAs, web browsers, etc.) with different styles and brand and across different locales.

In some embodiments of the KTP architecture, interfacing between the view and model layers (typically the most technically sophisticated and daunting component of a multi-tier architecture for non-programmers) is consolidated and encapsulated into easily reusable widgets.

Presentation for widgets is managed external to the widget with templates in some embodiments. This allows graphics designers, information architects, and web developers to focus on creating compelling customer experiences, and not the nitty gritty of multi-tier application development.

For widgets that are intended to expose dynamic content, the widget definition and the template are sufficient. Widgets that are intended to drive application functionality require collaboration with the Kinzan State Manager for the controller tier. These widgets typically have their own wizards and pipeline components, although they may certainly work with other wizards and pipeline components.

Widgets may also contain other widgets, allowing developers to present components at the appropriate level of granularity to page designers. For instance, a shopping basket widget may include a couple address widgets, several line item widgets, and a shipping and tax calculator widget. In this way, page designers have the flexibility to either use an off-the-shelf shopping basket widget, or ask that a more specialized shopping basket widget be made by assembling and rearranging the more atomic widgets in a different way.

As with all the various other modular artifacts in the KSP framework, widgets can be inherited and shared between different sites and applications, encouraging true reuse and consistency when assembling (versus building) applications.

The use of the Kinzan State Manager to apply a similar modular assembly approach to the controller tier is described in the "Kinzan State Manager Developer's Guide".

This document introduces the basics of the widget component model and development environment, including several examples and embodiments designed to highlight progressively more sophisticated elements of the widget component model.

3 Widget Overview

3.1 What Is A Widget?

A widget, in one embodiment, is a reusable component that may be combined with other widgets to form a page. A widget is used to represent a visual element on the page such as a paragraph or an image. A widget may also serve as a container to other widgets. A container (such as the page widget) relies on its children widgets for visual output.

The development of a widget typically involves declaring the widget in a Kinzan Application (kapp) file, creating a KCO file as a template to represent the presentation of the widget, implementing a set of Java classes (that conform to the Widget API) to manage the widget, and if necessary, implementing a wizard and pipeline components to support management of the widget using the Kinzan State Manager.

Once a widget is defined and available to a site, page developers may access the functionality of the widget by inserting `<widget type="widgetName">` tags in their HTML and KCOs. These widgets are configured with properties appropriate to their widget type.

Typically, page developers have access to an extensive library of widgets (represented by a tag library). Examples of possible widget types include `Text`, `Image`, `Banner`, `NavBar`, `StockQuote`, `SyndicatedContent`, `ShoppingBasket`, etc.

3.2 Advantage Of Using Widgets

Widgets have many advantages over conventional tag libraries, primarily because of the advantages of the KTP Framework. Specifically, widgets inherit runtime styling, layout, branding (assets), and localization, creating more opportunities for reuse. In addition, widgets integrate with the Kinzan State Manager, enabling them to respond to and drive events, and simplifying routing and processing of forms data when many widgets may be on a page. Finally, widgets are capable of supporting multiple target devices (such as wireless phones, PDAs, web browsers, etc.), allowing developers to share widget application and business logic across many device types by simply defining new presentation templates that are appropriate to the target device type.

Because widgets are resources within the KTP Framework, they also inherit useful features by being part of a site hierarchy. Specifically, widgets defined for a given site can be made visible and available to all children sites (note: children sites may override the definition of a widget from a parent site). This greatly facilitates the creation, management, and maintenance of reusable widget libraries across many sites.

Within a given site, widgets may be created at either the local page level or the site level. By defining widget instances in the kapp file for the site, the same widget instance is available to all pages in the site (note: a page may override the site definition of a widget instance). This allows central management and maintenance of site-wide resources. Examples may include a copyright notice, corporate branding and logo work, etc.

3.3 Components Of A Widget

3.3.1 Widget Factories

In one embodiment, widget factories create all widgets. The widget factory is an implementation of the factory pattern. The Rendering Service uses the widget factory to instantiate a widget without knowing which type of widget is being created or what steps are required to create the widget. The Rendering Service manages the widget factory. The Rendering Service creates and initializes the widget factory once during the first request for a particular widget. The Rendering Service uses the same widget factory for any subsequent requests for the same widget type.

Widget factories are made available to a site in the kapp file as follows:

```
<widgetFactoryList>
  .
  .
    <widgetFactory name="widgetFactoryName">
       <class>widgetFactoryClassname</class>
       <widgetTemplate>
          <variant target="HTML">WidgetTemplate.kco/>
       </widgetTemplate>
       <property name="configName1" value="configValue1"/>
       <property name="configName2" value="configValue2"/>
    </widgetFactory>
  .
  .
</widgetFactoryList>
```

A `GenericWidgetFactory` class is available to widget developers. For simple widgets, this factory is adequate. For more complex widgets, developers would extend this class to add in widget-specific functionality. This should only be necessary if the widget requires dynamic information to be share across all instances of the widget (for instance, shared caching of external data).

The most common case is requiring a widget factory that instantiates a custom class for the widget instances. This is supported by the `GenericWidgetFactory` class by setting the `WIDGET_CLASS` property with the class name for the required widget class.

3.3.2 Widget Instances

The widget factory is responsible for creating an instance of a widget class to manage the widget instance. Widget instances may be defined as either a local instance or a reference to a site-wide (or shared) instance. The syntax of defining a widget instance is equivalent across all files (KSP, KCO, etc.).

For a local widget instance, the widget factory is referenced in the KSPs and KCOs. The syntax for defining a local widget instance is as follows:

```
<widget type="widgetFactoryName">
   <property name="widgetInstanceConfigName1" value="widgetInstanceConfigValue1"/>
   <property name="widgetInstanceConfigName2" value="widgetInstanceConfigValue2"/>
</widget>
```

For a site-wide or shared widget instance, the instance must be defined in the kapp file for the site before other objects may reference this widget instance via the *sharedWidgetInstanceName*.

The syntax for defining a shared widget instance in the kapp file is as follows:

```
<widgetList>
  .
  .
<widget type="widgetFactoryName" name="sharedWidgetInstanceName">
   <property name="widgetInstanceConfigName1" value="widgetInstanceConfigValue1"/>
   <property name="widgetInstanceConfigName2" value="widgetInstanceConfigValue2"/>
</widget>
  .
  .
</widgetList>
```

This shared widget instance may then be referenced in KSPs and KCOs as follows:

```
<widget type="ref" name="sharedWidgetInstanceName"/>
```

Because the instance is shared at the site level, any changes to the shared widget are immediately reflected in the rest of the site.

A `GenericWidget` class is available to widget developers as the default class for widget instances. For simple widgets, this class is adequate. For more complex widgets, developers would extend this class to add in widget-specific functionality. This should only be necessary if the widget requires dynamic information that can not be managed with widget properties (for instance, unique methods to interface with the model layer).

Widget classes are instantiated by the Rendering Service every time a widget is rendered to a page. As such, all widget configuration should happen in the `init` method, with accessor methods that can be used from within the widget template KCO to get access to dynamic information.

3.3.3 Widget Properties

The `property` declarations in the widget and widget factory configurations define properties for the widget (instance and factory) and are widget dependent. Properties defined by a widget instance take precedence over properties defined by the widget factory, unless the widget factory has a visibility of "final".

A common technique is to define the default presentation for a widget at the factory level, and override the presentation for a particular instance by overriding the widget template. This allows page developers to modify widget presentation for each instance, while still benefiting from reuse of the widget business and application logic.

Widgets usually have properties that are unique to their implementation. However, the following properties are reserved for all widgets. The Rendering Service at runtime uses these properties.

- Template (required, set with the `<widgetTemplate>` tag): This property must define the different variants of the KCO that represents the presentation for the widget.
- `JSFILE` (optional): This property must contain the full path to the JavaScript file that should be included in the header portion of the HTML page. The JavaScript file should wrap its contents with the `<script>` tag.

3.3.4 Widget Templates

All widgets have a template that determines the KCO to be used to drive presentation of the widget. This KCO is processed and compiled into a servlet that coordinates with the widget factory and widget instance via a provided `WidgetProxy` class.

As a KCO, presentation is a JSP fragment that is dynamically styled by the runtime environment. As such, developers may use CSS style classes to drive the server-side styling of the widget presentation.

As a KCO, the widget KCO may also include references to other widgets and to named assets. Like other KCO assets, assets may have variants that are bound to style, device target, locale, and/or structure. The runtime environment will dynamically resolve use of the appropriate variant.

Java embedded within the KCO (recall that KCOs are JSP fragments that are processed by the Rendering Service) is used to drive all other presentation logic.

3.3.5 Nested Widgets

Widgets may include other widgets and reference them from within their widget template KCOs. This is done by declaring a `<widgetIncludeList>` when defining the widget. The syntax is as follows:

```
<widgetIncludeList>
  .
  .
    <widget type="incWidgetFactoryName1" binding="bindingName1">
       <property name="incPropertyName1" value="incPropertyValue1"/>
       <property name="incPropertyName2" value="incPropertyValue2"/>
    </widget>
    <widget type="incWidgetFactoryName2" binding="bindingName2">
       <property name="incPropertyName1" value="incPropertyValue1"/>
       <property name="incPropertyName2" value="incPropertyValue2"/>
    </widget>
  .
  .
</widgetIncludeList>
```

Included widgets are defined the same as all other widgets, with the exception of having an additional binding attribute that defines the binding name for the included widget. This binding name is used to refer to included widgets inside widget template KCOs.

To use included widgets inside widget template KCOs, the following must be included at the top of the KCO:

```
<widgetInclude binding="bindingName" action="import">
```

This creates a widget proxy (with variable name *bindingName*) that is bound to the included widget. It may be used within the widget template KCO like any other widget proxy. For example:

```
<%= bindingName.getWidget().getProperty( "incPropertyName1" ) %>
```

The rendering of an included widget within a widget template KCO is done as follows:

```
<widgetInclude binding="bindingName" action="display">
```

Widgets may include zero to many other widgets, and may be nested to many levels. It is typical for parent widgets to have a documented contract for coordinating with included widgets.

The stock quote widget example demonstrates the use of a generic panel widget with an included stock quote widget.

4 Example Widget Implementations

This section provides four example implementations of various embodiments of new widgets.

The first example involves creating a simple banner widget. The goal is to introduce you to the main concepts involved creating your own widget, as well as how you may utilize your new widget in your own KSPs. It also shows how a widget may support both HTML and WML (e.g., web phones) by having different presentations for each target device.

The second example involves a more complicated widget that is capable of downloading syndicated content in XML format, applying an XSL transformation, and displaying the results. This example is intended to demonstrate how to use and extend some core widget classes and exposes you to more of the widget API.

The third example shows how to integrate widgets with the Kinzan State Manager so that input from widgets can drive applications and change widget behavior. Specifically, the simple weather widget is able to manage the zip code for which it is to display weather information.

The fourth example involves integrating a widget with an external service on the back end, and embedding it within a standard GUI widget on the front end. Specifically, the stock quote widget interfaces with a stock quote service (which retrieves and caches up to date stock quote information from the internet using XML). It also interfaces with a generic panel widget that exposes different modes in the widget, allowing the user to input the list of stocks that they are interested in. Personalization information is tracked using Enterprise Java Beans (EJBs), based on a user name that is provided by a simple login widget.

4.1 Banner Widget

The banner widget allows the page designer to place a banner and its associated URL on a page. The page designer should be able to configure the banner image, the URL to link to when the banner is clicked, and other image attributes such as the width, height, etc. The widget should support both traditional web browsers with HTML, as well as support wireless Internet access with Wireless Markup Language (WML).

As presented, this is a relatively trivial (and somewhat contrived) application of a widget to a problem that may best be addressed directly with HTML or WML. However, it does introduce key concepts and provides the foundation for implementing more sophisticated banner widgets that may include support for localization, personalization, manage banner impressions, and provide targeted banners based on page content and/or user information.

The following list shows the configurable widget properties we will build into our simple example:

Property	Required	Description
URL	Yes	The URL to be associated with the banner
IMAGE	Yes	The path relative from the web document root where the image is.
ALT	Yes	Text to be displayed in place of the image for character based browsers or on wireless devices.
BORDER	Optional	The thickness of the border around an image element
WIDTH	Optional	The horizontal length in percentage or pixels that should be reserved for the

		image.
HEIGHT	Optional	The vertical length in percentage or pixels that should be reserved for the image.

4.1.1 Banner Widget Factory Declaration

To make the widget available for use, the banner widget must be declared in the kapp file for the site. The following excerpt shows how this is done:

```
1   <widgetFactoryList>
    .
    .
2      <widgetFactory name="banner" visibility="public">
3         <class>net.kinzan.rendering.component.GenericWidgetFactory</class>
4         <description>Banner Widget</description>
5         <widgetTemplate>
6            <variant target="HTML">bannerTemplate.kco</variant>
7            <variant target="WML">wml/bannerWMLTemplate.kco</variant>
8         </widgetTemplate>
9         <property name="ALT" value="[LINK]"/>
10     </widgetFactory>
    .
    .
11  </widgetFactoryList>
```

1 The widget factory must be declared in the widget factory list section of the kapp file.

2 Specify the name of the widget factory. This is the name that should be used to obtain the widget proxy in the widget KCO file.

3 Specify the Java class name that should be used to instantiate the widget factory. In this case, the `GenericWidgetFactory` class is used.

6 Declare the widget template KCO for the widget when the target device is HTML (in this case `bannerTemplate.kco`).

7 Declare the KCO file to be used when the target device is a WML device. By convention, target variants are managed in the *`assetType/targetVariant`* directory within the appropriate directory. In this case, the KCO file to be used when rendering to WML targets is in the `wml` directory within the `kco` directory.

9 Set the default value of the `ALT` property to be the string `[LINK]`.

4.1.2 Banner Widget Template KCO File Implementations

Since there is no special behavior that needs to be implemented outside the widget KCO file, the banner widget uses the `GenericWidget` and the `GenericWidgetFactory` classes.

4.1.2.1 Widget Template KCO File for HTML Targets

The implementation of the banner widget template KCO file for HTML targets (in this case, a KCO file called `bannerTemplate.kco`) is as follows:

```
1   <%@ page import="net.kinzan.rendering.Widget"%>
2
3   <%Widget myWidget = banner.getWidget(); %>
4
```

```
5   <a href="<%=myWidget.getProperty( "URL" )%>">
6      <img src="<%=myWidget.getProperty( "IMAGE" )%>"
7         <% if ( myWidget.getProperty( "BORDER" ) != null )
8               out.print( "border=\"" + myWidget.getProperty( "BORDER" ) + "\" "
    );

9           if ( myWidget.getProperty( "WIDTH" ) != null )
10             out.print( "width=\"" + myWidget.getProperty( "WIDTH" ) + "\" " );

11          if ( myWidget.getProperty( "HEIGHT" ) != null )
12             out.print( "height=\"" + myWidget.getProperty( "HEIGHT" ) + "\" "
    );

13          if ( myWidget.getProperty( "ALT" ) != null )
14             out.print( "alt=\"" + myWidget.getProperty( "ALT" ) + "\" " );
15       %>
16    >
17  </a>
```

3 At the request scope, a widget proxy for the widget named `banner` is automatically instantiated. Note that the widget proxy instance name matches the widget factory instance name specified in the kapp file. This is used to obtain a reference to the widget from the widget proxy and assign it to `myWidget`.

5-15 Use the `getProperty` method to obtain the banner widget properties and complete the HTML code.

4.1.2.2 Widget Template KCO File for WML Targets

The implementation of the banner widget template KCO file for WML targets (in this case, a KCO file called `bannerWMLTemplate.kco`) is as follows:

```
1   <%@ page import="net.kinzan.rendering.Widget"%>
2
3   <%Widget myWidget = banner.getWidget(); %>
4
5   <card>
6      <p align="center">
7         <a href="<%=myWidget.getProperty( "URL" )%>">
8         <% if ( myWidget.getProperty( "ALT" ) != null )
9               out.print(myWidget.getProperty( "ALT" ));
10        %>
11        </a>
12     </p>
13  </card>
```

3 At the request scope, a widget proxy for the widget named `banner` is automatically instantiated. Note that the widget proxy instance name matches the widget factory instance name specified in the kapp file. This is used to obtain a reference to the widget from the widget proxy and assign it to `myWidget`.

4 Obtain a reference to the widget from the widget proxy and assign it to `myWidget`.

7-11 Use the `getProperty` method to obtain the banner widget properties and complete the WML code. Note that this presentation ignores the banner image, but is still part of the same banner widget.

4.1.3 Sample Page

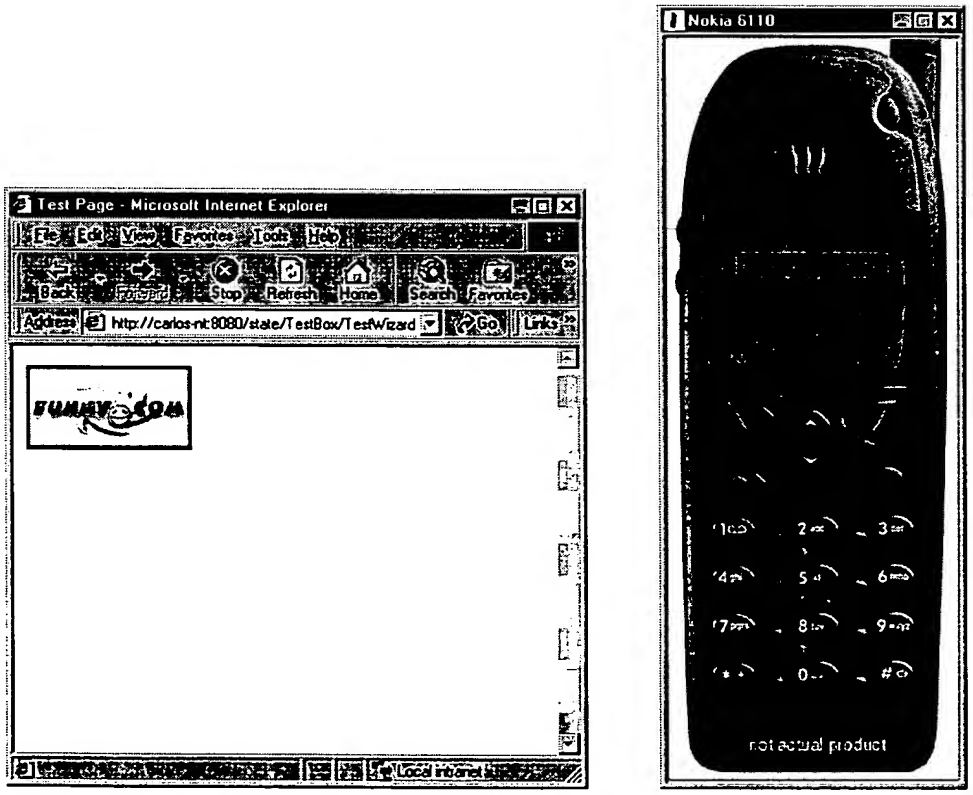
Once all the above are declared in the kapp file for the site, the banner widget may be used as follows:

```
1  <widget type="banner">
2     <property name="IMAGE"
                value="http://www.funny.com/linkImages/zigzagFunny.gif"/>
3     <property name="URL" value="http://www.funny.com"/>
4     <property name="ALT" value="Click here for grins!"/>
5  </widget>
```

1 Instantiate a new instance of a banner widget.

2-4 Set the `IMAGE`, `URL`, and `ALT` properties for the new widget instance. Note that the `<widgetTemplate>` parameter default that was specified with the widget factory may be overridden if a different presentation is required.

The result (when run with the TestBox test scaffold provided with the examples) is shown below for both an HTML target hitting the web site and a WML target hitting the web site (in this case, the Nokia WAP phone simulator).



4.2 Simple Syndicated Content Widget

The simple syndicated content widget pulls information from a user-specified site and displays it on the page area assigned to the widget. The information is retrieved in XML format. The widget uses XSL to transform the XML data into an HTML fragment for display. The HTML fragment for the widget is then available for rendering.

Since the content is retrieved from a remote site every time the page is requested, the widget factory caches the XML data. To prevent outdated content, the data in the cache is flushed at a user-specified interval.

This example demonstrates linking to external XML sources and using XSL to manage presentation (instead of using a widget template KCO for all presentation, which is the preferred technique). It is intended to demonstrate the use of a custom widget factory and a custom widget instance Java classes. It can also be easily generalized to support XML+XSL transformation for any XML source, not just syndicated content.

The more advanced stock quote widget (presented in section 4.4) demonstrates the preferred method of encapsulating external XML sources within a Java service, which can then be easily integrated with a KCO to allow flexible presentation. In addition, an enhanced version of a syndicated content widget that uses a syndicated content service and manages all presentation with a KCO is part of the MyPortal.com example application.

These are the configurable properties for the simple syndicated content widget:

Property	Required	Description
URL	Yes	The URL used to fetch the XML data.
XSL	Yes	The absolute path where the XSL file can be found.
AGING_INTERVAL	Yes	The maximum time (in seconds) that entries in the data cache should be maintained.
TITLE	Yes	This is the title that will be displayed for the widget.

Since XSL basically drives all the presentation, most of the widget functionality is implemented in the `SimpleSyndicatedContentWidgetFactory` and `SimpleSyndicatedContentWidget` classes.

4.2.1 Simple Syndicated Content Widget Factory Declaration

The simple syndicated content widget factory declaration is similar to the banner widget factory:

```
1   <widgetFactoryList>
    .
    .
2      <widgetFactory name="simpleSyndicatedContent">
3
    <class>com.kinzan.example.simplesyndicatedcontent.widget.SimpleSyndicatedConten
    tWidgetFactory</class>
4          <description>Simple Syndicated Content Widget</description>
5          <widgetTemplate>
6             <variant target="HTML">simpleSyndicatedContentTemplate.kco</variant>
7          </widgetTemplate>
6          <property name="AGING_INTERVAL" value="600"/>
7      </widgetFactory>
    .

8   </widgetFactoryList>
```

3 The class element is set to the `SimpleSyndicatedContentWidgetFactory` class.

6 The aging interval property is declared to have a default value of 600 seconds.

4.2.2 Simple Syndicated Content Widget Factory

`SimpleSyndicatedContentWidgetFactory` inherits its default behavior from `GenericWidgetFactory` (see Appendix or example code for full listing).

Since the widget factory is persistent over page requests, the content cache is maintained in the widget factory. The content cache is initialized in the `init` method, which gets invoked after the widget factory is created

```
1  private WeakCache iContentCache = null;

2  public void init( AppContext appContext,
                     WidgetFactoryDescriptor factoryDescriptor )
       throws InitWidgetFactoryException
3  {
4     super.init( appContext, factoryDescriptor );

5     long maxCacheAge = Long.parseLong( getProperty( "AGING_INTERVAL" ) );
6     iContentCache = new WeakCache( new AgingCachePolicy( maxCacheAge ) );

7  }
```

1 The simple syndicated content widget factory redefines the `init` method inherited from `GenericWidgetFactory`.

4 Call the generic widget factory `init` method to get the default behavior.

5 Get the maximum age of a cache entry. This parameter was defined in the kapp file when the factory was declared.

6 Create the content cache and assign it to the private data holder `iContentCache`. The `WeakCache` class implements a cache based on the policy specified in the constructor. The `WeakCache` class is a utility class in the Kinzan library. For more information please see the corresponding class documentation.

`SimpleSyndicatedContentWidgetFactory` class is also responsible for instantiating new widgets. The `SimpleSyndicatedContentWidgetFactory` class overrides the `createWidget` method to create and initialize an instance of `SimpleSyndicatedContentWidget`.

```
1  public Widget createWidget( RequestContext context, WidgetDescriptor wd )
       throws CreateWidgetException, InitWidgetException
2  {
3     SimpleSyndicatedContentWidget sscWidget = null;
4     String xmlData = null;

5     try
6     {
7        sscWidget = new SimpleSyndicatedContentWidget( wd, this );

8        sscWidget.init( context, wd, this );

9        String xmlURL = sscWidget.getProperty( "URL" );
10       String xmlData = (String) iContentCache.get( xmlURL );

11       if ( xmlData == null )
12       {
```

```
13          URL url = new URL( xmlURL );
14          InputStream inStream = url.openStream();

15          byte[] buffer = new byte[1024];
16          StringBuffer tempString = new StringBuffer();

17          int count = 0;
18          while ( (count = inStream.read( buffer ) ) != -1)
19          {
20             tempString.append( new String( buffer, 0, count ) );
21          }

22          xmlData = tempString.toString();
23          inStream.close();

24          if ( xmlData != null )
25          {
26             iContentCache.put( xmlURL, xmlData );
27          }
28       }
29       if ( xmlData != null )
30          sscWidget.processXMLData( xmlData );
31       else
32          sscWidget.setContent( "Data not available!" );
33    }

34    catch( Exception e )
35    {
36       throw new CreateWidgetException( e.getMessage() );
37    }

38    return sscWidget;
39 }
```

7 Since `SimpleSyndicatedContentWidget` is derived from the `Generic` widget, the constructor must take a widget descriptor object and a widget factory.

9-10 Get the URL for the XML data from the widget.

10-28 Using the URL as a key, check if the XML data is in the cache. If it is not in the cache, get the XML data and place it in the cache. Note that data in the cache will be removed when the system is low on memory or the data has expired.

29-32 Call the widget to transform the XML data to HTML.

4.2.3 Simple Syndicated Content Widget Class

The `SimpleSyndicatedContentWidget` class is derived from the `GenericWidget` class. In addition to the normal widget functions, it needs to transforms the raw XML data to HTML.

To do this, the `SimpleSyndicatedContentWidget` defines two additional methods: `getContent` and `processXMLData`. The `getContent` method is a simple accessor to the processed XML data. The `processXMLData` method is implemented as follows:

```
1  public void processXMLData( String xmlData )
2     throws IOException, SAXException
3  {

4     FileReader xslFile = new FileReader( getProperty( "XSL" ) );
5     StringReader xmlSource =  new StringReader(xmlData );
```

```
6      StringWriter output = new StringWriter( 2048 );

7      XSLTProcessor processor = XSLTProcessorFactory.getProcessor();
8      processor.process( new XSLTInputSource( xmlSource ),
9                         new XSLTInputSource( xslFile ),
10                        new XSLTResultTarget( output ) );

11     iContent = output.getBuffer().toString();

12 }
```

1 The XML data is passed to the `processXMLData` method. Because we want to cache the XML content for all instances of the simple syndicated content widget, the widget factory is responsible to fetching the XML content and passing it to this method.

4 Use the `getProperty` method to get the XSL file name from of the `XSL` property for the widget, then read the XSL file.

5 Create a String buffer out of the XML data.

6 Create a String output buffer to save the output of the transformation.

7-10 Call the XSL processor to transform the XML data to an HTML fragment.

11 Save the HTML fragment in a private data member, which is accessible via the `getContent` method.

4.2.4 Simple Syndicated Content Widget Template KCO File Implementation

The widget template KCO file references the `SyndicatedContentWidget` instance to place the HTML fragment in a table:

```
1  <%@ page
   import="com.kinzan.example.simplesyndicatedcontent.widget.SimpleSyndicatedConte
   ntWidget"%>
2
3  <%SimpleSyndicatedContentWidget myWidget =
        (SimpleSyndicatedContentWidget) simpleSyndicatedContent.getWidget(); %>
4
5  <table>
6     <tr>
7        <th><%=myWidget.getProperty( "TITLE" )%></th>
8     </tr>
9     <tr>
10       <td><%=myWidget.getContent()%></td>
11    </tr>
12 </table>
```

4 Get the widget via the widget proxy and cast the widget to a `SimpleSyndicatedContentWidget` type.

7 Use the `TITLE` property to put the heading on the table.

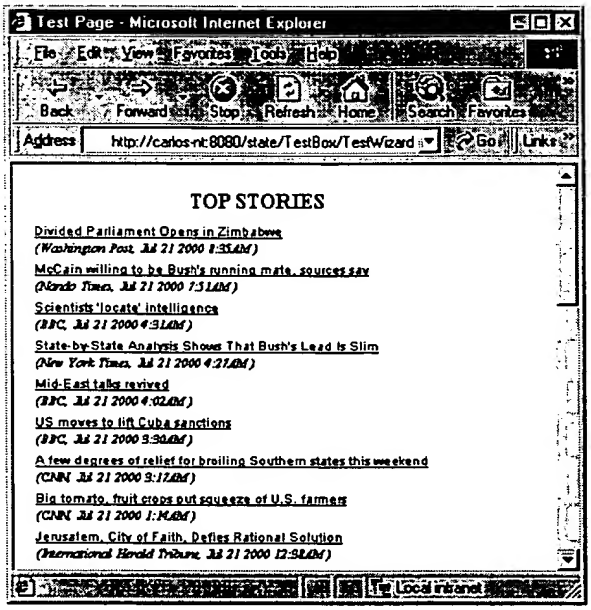
10 Get the HTML content from the widget and insert it in the table.

4.2.5 Sample Page

This sample shows how the syndicated content widget may be used to pull data from the moreover.com "Top Stories" web site

```
1  <widget type="simpleSyndicatedContent">
2     <property name="URL"
            value="http://p.moreover.com/cgi-local/page?index_topstories+xml"/>
3     <property name="XSL"
   value="e:/projects/examples/TestBox/deployment/webroot/TestBox/resource/xsl/mor
   eover.xsl"/>[1]
4     <property name="TITLE" value="TOP STORIES"/>
5  </widget>
```

In this case, the XSL file was written to take the particular XML provided by moreover.com and transform it into the HTML presentation shown in the figure below. The XSL file can be found below, but is specific to this particular presentation and the XML feed from moreover.com.



In general, the preferred method for XML integration is to wrap the XML data source with a Java service using the Kinzan Services Manager. This service can then be queried from within a KCO to give flexible presentation and styling. This better leverages the power of the KTP Framework to avoid needing to write a new XSL file every time the style or presentation of a page needs to be tweaked.

4.3 Simple Weather Widget

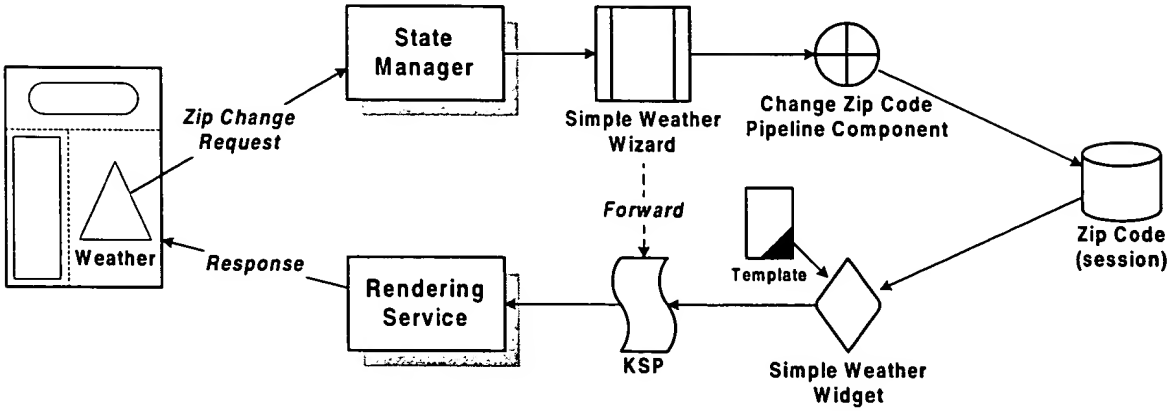
While the previous examples demonstrated the use of widgets to display dynamic information on a page, this example demonstrates how widgets can be used in combination with the Kinzan State Manager to process user input to configure a widget.

Given a zip code, the simple weather widget retrieves the weather information and displays it on a web page. The information is returned as a GIF image from http://weatherguys.com/zipcode/*X*/*YYYYY*.gif, where *X* is the first digit in the zip code, and *YYYYY* is the five-digit zip code.

It also allows a user to enter a zip code, which is stored in the user's session for when the page gets updated.

[1] This simplistic widget uses a hard-coded path to the XSL file. This was done to keep the example simple. A more general solution would be to retrieve the XSL file via an URL relative to the document root of the web server. The XSL file could then be managed in the `/resource` directory for the site.

Here is an overview of how the various pieces fit into a typical request/response loop:



The simple weather widget provides a form field where the user may input a zip code. The action of the form is to submit various form variables to the State Manager (see the "Kinzan State Manager Developer's Guide"). The State Manager then looks up the appropriate action for a request to change a zip code, invokes a pipeline component that saves the zip code in the user's session, and forwards to the KSP page that originally submitted the request to the Rendering Service. When the Rendering Service goes to render the original page, the simple weather widget queries the session to retrieve the new zip code for the weather information and constructs the appropriate URL to retrieve the weather information.

While sessions are one technique to manage user-based information, it does not work when the same personalization information needs to be available in multiple sites or across multiple devices. The MyPortal.com example application includes an enhanced version of a weather widget that is integrated with a common login and personalization system to allow server-based persistence of personalization information.

These are the configurable properties for the simple weather widget:

Property	Required	Description
URL	Yes	The base URL for where the weather data is to be retrieved.
ZIP_CODE	Yes	The zip code to be used to retrieve weather data.
WIDTH	Optional	The horizontal length in percentage or pixels that should be reserved for the image.
HEIGHT	Optional	The vertical length in percentage or pixels that should be reserved for the image.

4.3.1 Simple Weather Widget Factory Declaration

The simple weather widget factory declaration is similar to the previous examples:

```
1   <widgetFactoryList>
    .
    .
2       <widgetFactory name="simpleWeather" visibility="public">
```

```
3         <class>net.kinzan.rendering.component.GenericWidgetFactory</class>
4         <description>Simple Weather Widget</description>
5         <widgetTemplate>
6            <variant target="HTML">simpleWeatherTemplate.kco</variant>
7         </widgetTemplate>
8         <property name="WIDGET_CLASS"
             value="com.kinzan.example.simpleweather.widget.SimpleWeatherWidget"/>
9         <property name="URL" value="http://weatherguys.com/zipcode/"/>
10        <property name="WIDTH" value="200"/>
11        <property name="HEIGHT" value="150"/>
12     </widgetFactory>

13  </widgetFactoryList>
```

3 The class element is set to the `GenericWidgetFactory` class.

5-7 The default template for the widget is set to `simpleWeatherTemplate.kco`.

8 The `WIDGET_CLASS` property is set to the `SimpleWeatherWidget` class. This property is used by the `GenericWidgetFactory` to instantiate the appropriate class for widget instances. The alternative is to create a custom widget factory for this widget that instantiates the appropriate widget class for widget instances, along with any other custom factory behavior this widget requires. However, since the only custom factory behavior is instantiating the appropriate custom widget class, the `WIDGET_CLASS` property of the `GenericWidgetFactory` is sufficient.

9 The default `URL` property is set to the base URL to retrieve the weather information.

10-11 The default `WIDTH` and `HEIGHT` properties are set to the image size returned by weatherguys.com.

4.3.2 Simple Weather Widget Class

The `SimpleWeatherWidget` class is derived from the `GenericWidget` class. It implements a `getURL` method that returns the URL for the appropriate weather GIF. The appropriate URL is determined in the `init` method for the class, which is invoked every time the widget is rendered. The method queries the session to see if a zip code has been entered. If not, the widget `ZIP_CODE` property is used to construct the URL.

The `init` method is implemented as follows:

```
1  private String iWeatherURL = "";

2  public void init( RequestContext context, WidgetDescriptor wd,
                     WidgetFactory factory )
      throws InitWidgetException
3  {
4     super.init( context, wd, factory );

5     String zipCode = "";
6     zipCode = (String)
         context.getSession().getAttribute("simpleWeather.zipCode");

7     if ( zipCode == null)
8        zipCode = getProperty("ZIP_CODE");

9     iWeatherURL = getProperty( "URL" ) + zipCode.charAt(0) + "/"
                  + zipCode + ".gif";
10 }
```

6 Attempt to retrieve the zip code from the session.

7-8 If the zip has not been set in the session, grab the `ZIP_CODE` property for the widget as the default.

9 Construct the URL for the weather GIF using the specification from weatherguys.com

The `getURL` method is implemented as follows:

```
1  public String getURL( )
2  {
3     return iWeatherURL
4  }
```

3 Return the URL as constructed in the `init` method.

4.3.3 Simple Weather Widget Template KCO File Implementation

The widget template KCO file references the `SimpleWeather` widget instance to link to the appropriate weather information, and has an input form for a new zip code that submits to simple weather wizard via the State Manager:

```
1  <%@ page import="net.kinzan.servlet.RequestContext"%>
2  <%@ page import="com.kinzan.example.simpleweather.widget.SimpleWeatherWidget"%>
3
4  <%RequestContext context =
              (RequestContext)request.getAttribute( "net.kinzan.requestContext" );
   %>
5  <%SimpleWeatherWidget myWidget =
              (SimpleWeatherWidget) simpleWeather.getWidget(); %>
6
7  <table>
8     <tr>
9        <th>WEATHER</th>
10    </tr>
11    <tr>
12       <td>
13          <img src="<%=myWidget.getURL()%>"
              width='<%=myWidget.getProperty("WIDTH")%>'
              height='<%=myWidget.getProperty("HEIGHT")%>' >
14       </td>
15    </tr>
16    <%
17      if (context.getError() != null && context.getError().getMessage() != null)
18      {
19     %>
20    <tr><td><%=context.getError().getMessage()%></td></tr>
21    <%
22      }
23     %>
24    <tr>
25       <td>
26          <form action="/state" method="POST">
27             <input type="HIDDEN" name="net_kinzan_requestContextID"
                                    value="<%= context.getID()%>">
28             <input type="HIDDEN" name="net_kinzan_nextWizard"
                                    value="SimpleWeatherWizard">
```

```
29            <input type="HIDDEN" name="net_kinzan_nextState"
                                   value="xChangeZipCode">
30            <table>
31               <tr>
32                  <td>
33                     Enter zip code:
34                  </td>
35               </tr>
36               <tr>
37                  <td>
38                     <input type="text" name="NEW_ZIP_CODE" value="">
39                  </td>
40               </tr>
41               <tr>
42                  <td>
43                     <input type="SUBMIT" value="Change Zip" >
44                  </td>
45               </tr>
46            </table>
47         </form>
48      </td>
49   </tr>
50 </table>
```

4 Get the request context from the request.

5 The `simpleWeather` widget proxy is automatically available to the KCO.

13 Use the simple weather widget `getURL` method and properties to specify the image source for the appropriate zip code (widget uses zip code in session to construct URL).

16-23 If an error has been posted to the request context (for instance, for a zip code in the incorrect format), output the error message.

27 Include the request context ID so the State Manager can properly process the form (required).

28-29 Set the appropriate form variables to direct the State Manager to invoke the `xChangeZipCode` state in the `SimpleWeatherWizard` wizard.

38 The name of the form variable ("`NEW_ZIP_CODE`") will be available to the `xChangeZipCode` state in the request context.

4.3.4 Simple Weather Wizard Implementation

The simple weather wizard follows:

```
1  <?xml version="1.0"?>

2  <!DOCTYPE wizard SYSTEM "http://www.kinzan.net/dtd/wizard.dtd">

3  <wizard name="SimpleWeatherWizard">
4     <actionState name="xChangeZipCode" component=
          "com.kinzan.example.simpleweather.component.ChangeSimpleWeatherZipCode">
5        <transition event="SUCCESS" state="END"/>
6        <transition event="FAILURE" state="END"/>
7     </actionState>

8     <actionState name="END"
         component="net.kinzan.state.pipeline.component.EndStateComponent"/>
9  </wizard>
```

3 Declare the name of the wizard.

4-7 Configure the `xChangeZipCode` action state (by convention, action states begin with "x") to use the `ChangeSimpleWeatherZipCode` pipeline component, and to transition to the `END` state when done.

8 Configure the `END` state to use the provided `EndStateComponent`, which returns control to the state that initially invoked the wizard. In this case, control is returned to the state that generated the change zip code request.

4.3.5 Change Zip Code Pipeline Component[2]

The new methods in `ChangeSimpleWeatherZipCode` class are as follows:

```
1  public ChangeSimpleWeatherZipCode()
   throws org.apache.regexp.RESyntaxException
2  {
3     super();

4     ComponentParameter zipCodeParameter =
         new ComponentParameter("NEW_ZIP_CODE");
5     zipCodeParameter.setNullOkay( false );

6     RegularExpressionValidator regExValidator =
         new RegularExpressionValidator( "^[0-9][0-9][0-9][0-9][0-9]$" );
7     zipCodeParameter.setParameterValidator( regExValidator );

8     iInputParameters.addComponentParameter( zipCodeParameter );
9  }

10 public String processEvent( RequestContext context )
   throws InvalidParameterException, TransactionAbortedException, TimedOutException
11 {
12    super.processEvent( context );

13    String zipCode = context.getProperty( "NEW_ZIP_CODE" );
14    context.getSession().setAttribute( "simpleWeather.zipCode", zipCode );

15    return SUCCESS;
16 }
```

1 The constructor is used to configure the automatic parameter validation that the pipeline component will be using.

4 Construct a validator for the `NEW_ZIP_CODE` parameter (passed to the pipeline component by the widget).

5 Configure the validator to throw an exception if the parameter is null.

6 Create a regular expression validation rule that only accepts 5 numbers.

---

[2] Note: Strictly speaking, this pipeline component is unnecessary for this example. The `NEW_ZIP_CODE` property is maintained in the request context so can be directly accessed by the widget without storing in the session. If the widget were to be modified to pull the `NEW_ZIP_CODE` property from the request context instead of the session, the wizard could be simplified to only the `END` state. However, this example can be more easily extended if the developer would likely want to manage persistence outside the session (for instance, in a database).

7 Configure the validator to throw an exception if the 5-digit validation rule is violated.

8 Add the parameter to the list of parameters to be automatically validated when the pipeline component is invoked.

10 The `processEvent` method is called by the State Manager when the pipeline component is invoked.

13 Get the `NEW_ZIP_CODE` property from the request context.

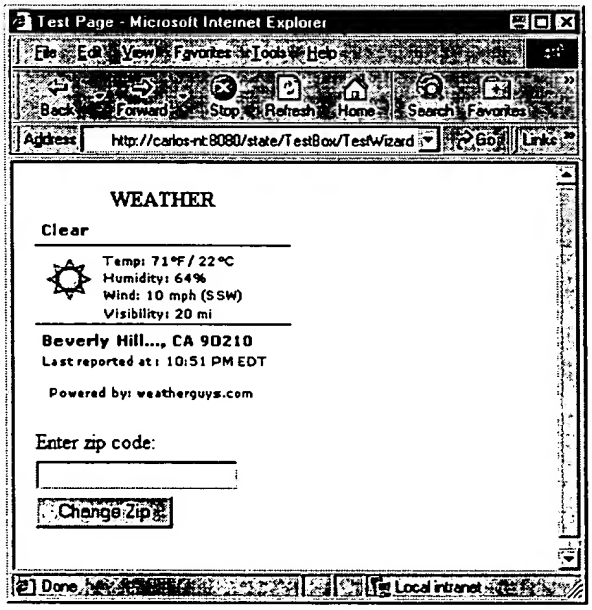
14 Store the new zip code in the session.

15 Return a `SUCCESS` event to drive the State Manager to the next state (in this case, the `END` state as defined in the wizard in the previous section).

4.3.6 Sample Page

This sample shows how the simple weather widget may be used on a page:

```
1  <widget type="SimpleWeather">
2     <property name="ZIP_CODE" value="90210"/>
3  </widget>
```



In general, personalization information should persist between user sessions and be available in different applications and from different devices.

4.4 Stock Quote Widget

This example demonstrates integration of a widget with the Kinzan State Manager and a back end service.

Given a list of stocks, the stock quote widget interrogates a stock quote service to retrieve stock quotes, and displays it on a web page. The service polls an Internet source for the financial data in XML format, and caches it within the service. It also manages periodically refreshing the data.

For authenticated users, the stock quote widget allows the user to enter a list of stocks they are interested in. This stock list for the user is managed via the stock quote service as Enterprise Java Beans (EJBs).

To enable personalization, the widget works with a simple login widget to provide extremely basic authentication.

The widget is able to support display and configuration to both HTML and WML targets, and the shared personalization information allows a users stock preferences to be available from within multiple applications.

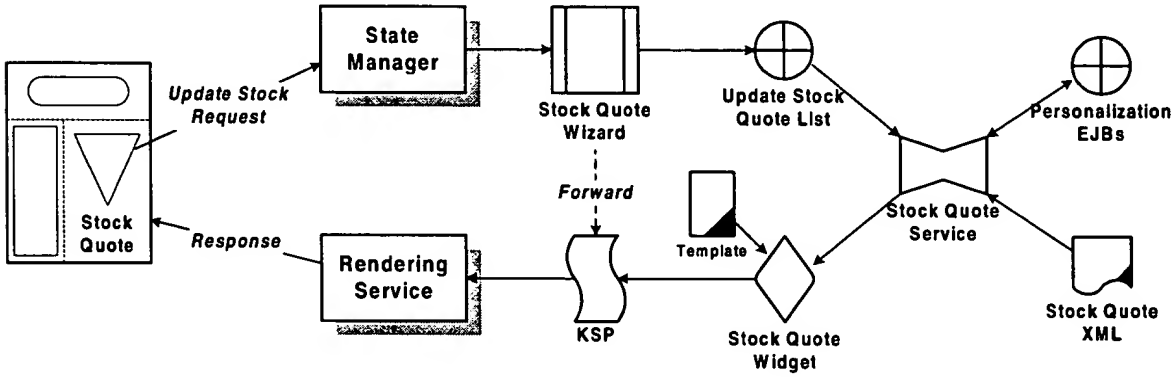
The widget allows all users to do spot lookups of stock quotes and displays the results along with the the users preferred stock list.

The widget has multiple modes, presenting differently when it is displaying stock quotes from when it is allowing the user to enter or update their list of stocks. The result is in-place editing when the widget is placed on a page.

The widget implements the `PanelContentWidget` interface, allowing it to be used within a panel widget from the Kinzan WebTop Framework. The panel widget coordinates with the stock quote widget to determine if the user is allowed to switch the widget to edit mode.

The combination of the simple login widget, panel widget, and stock quote widget present an opportunity to demonstrate widgets being included in other widgets, and inter-widget communication.

Here is an overview of how the various pieces fit into a typical request/response loop:



The stock quote widget displays a list of stocks and provides a form field where the user may input a any stock symbol to lookup. The action of the form is to put the requested symbol in the request context, and forward to the KSP to redisplay. The stock quote widget then appends the quote for the lookup symbol at the end of the list of stocks (default stocks, or stocks specific to the user if the user has logged in).

If a user has logged in, the widget uses the user's user ID as the primary key to query the stock quote service for the list of preferred stock symbols for that user. The stock quote service uses this key as the primary key to look up the appropriate personalization EJB. The EJB interface is encapsulated in the stock quote service, so the widget developer need only deal with the simple stock quote service API.

Because the widget implements the `PanelContentWidget` interface, it can be included in a panel widget. The panel widget interrogates the stock widget for the title to display at the top of the panel. In addition, if the user has logged in, the panel widget displays an edit link that messages the State Manager to switch the stock widget to `EDIT` mode.

In `EDIT` mode, the user is able to change their list of preferred stocks. When the stock-list form is submitted, a pipeline component passes the stock list to the stock quote service for persistence. If the user does not exist in the stock quote service, the stock quote service automatically creates personalization EJBs for the user and stores the user's stock list. The mode of the widget is changed

back to DISPLAY mode, so quotes for the revised stock list are displayed next time the widget is rendered.

These are the configurable properties for the stock quote widget:

Property	Required	Description
STOCK_QUOTE_SERVICE	Yes	Name of the stock quote service the widget and pipeline components will connect to.
DEFAULT_SYMBOLS	Optional	Comma delimited list of stock symbols to display by default for non-authenticated users.
EDIT_WIZARD	Yes	Name of the wizard to be invoked when the edit button is pressed on a panel widget that includes the stock quote widget.
EDIT_STATE	Yes	Name of the state to be invoked when the edit button is pressed on a panel widget that includes the stock quote widget.
WIDTH	Optional	The width (in pixels or percent) for the panel widget that includes the stock quote widget.

4.4.1 Stock Quote Widget Factory Declaration

To make the widget available for use, the stock quote widget factory must be declared in the kapp file for the site. The following shows how this is done:

```
1   <widgetFactoryList>
      .
      .
2       <widgetFactory name="stockQuote" visibility="public">
3          <class>net.kinzan.rendering.component.GenericWidgetFactory</class>
4          <description>StockQuote Widget </description>
5          <widgetTemplate>
6             <variant target="HTML">stockQuoteTemplate.kco</variant>
7          </widgetTemplate>
8          <property name="WIDGET_CLASS"
                     value="com.kinzan.example.stockquote.widget.StockQuoteWidget"/>
9          <property name="STOCK_QUOTE_SERVICE" value="StockQuoteService"/>
10         <property name="DEFAULT_SYMBOLS" value="^DJI,^IXIC,^XAX"/>
11         <property name="EDIT_WIZARD" value="StockQuoteWizard"/>
12         <property name="EDIT_STATE" value="xSetEditMode"/>
13         <property name="WIDTH" value="300"/>
14      </widgetFactory>
      .
      .
15  </widgetFactoryList>
```

* Required to work with PanelContentWidget interface.

3 The class element is set to the `GenericWidgetFactory` class.

5-7 The default template for the widget is set to `stockQuoteTemplate.kco`.

8 The `WIDGET_CLASS` property is set to the `StockQuoteWidget` class, which will be used by the `GenericWidgetFactory` to instantiate the appropriate class for widget instances.

9 The default name of the stock quote service is set to `StockQuoteService`.

10 The default set of stock symbols is set to a comma-delimited list of the symbols for the composite indices of the major stock exchanges (Dow Jones, NASDAQ, etc.).

11-13 Properties needed by the panel widget.

4.4.2 Stock Quote Widget Class

The `StockQuoteWidget` class is derived from the `GenericPanelContentWidget` class, allowing it to be included within a generic panel widget.

It implements a `getSymbols` method that returns the list of active symbols for the widget. The appropriate symbol list is constructed in the `init` method for the class, which is invoked every time the widget is rendered. It also implements a `getStockQuotesList` that returns a linked list of stock quotes for the active symbols for the widget. This list is also constructed in the `init` method.

In support of the `PanelContentWidget` interface, the widget also implements a `getTitle` method that returns the title that should be displayed at the top of the panel widget, and an `isEditable` method that returns "true" if the panel widget is supposed to activate an edit button that will change the stock quote widget from `DISPLAY` mode to `EDIT` mode. It also implements `editWizard` and `editState` methods that are used by the panel widget to construct the appropriate request to change the stock quote widget into `EDIT` mode.

The `init` method is implemented as follows:

```
1   private String iWidgetMode = null;
2   private String iUser = null;
3   private String iSymbols = null;
4   private String iContextID = null;
4   private WidgetDescriptor iWd = null;
5   private LinkedList iStockQuotesList = null;

6   public void init( RequestContext context, WidgetDescriptor wd,
                      WidgetFactory factory )
       throws InitWidgetException
7   {

8      super.init( context, wd, factory );
       iContextID = context.getID();
9      iWd = wd;

10     try
11     {

12        String stockQuoteServiceName = getProperty( "STOCK_QUOTE_SERVICE" );
13        context.setProperty( "net.kinzan.stockQuoteServiceName",
                               stockQuoteServiceName );

14        StockQuoteService stockQuoteService =
             (StockQuoteService) context.getService( stockQuoteServiceName );
```

```
15        iWidgetMode = context.getProperty( "net.kinzan." +
                                             wd.getWidgetDescriptorId() +
                                            ".widgetMode" );
16        if ( iWidgetMode == null )
17           iWidgetMode = "DISPLAY"; // Set to DISPLAY mode by default

18        String coreSymbols = null;

19        iUser = (String) context.getSession().getAttribute( "simpleLogin.user" );

20        if ( iUser != null && stockQuoteService.isUserRegistered( iUser ) )
21        {
22            coreSymbols = stockQuoteService.getSymbolByUser( iUser );
23        }
24        else
25        {
26           coreSymbols = getProperty( "DEFAULT_SYMBOLS" );
27        }

28        String lookupSymbol = null;

29        if ( !"EDIT".equals( iWidgetMode ) )
30        {
31           lookupSymbol = context.getProperty( "LOOKUP_SYMBOL" );
32        }

33        iSymbols = coreSymbols;

34        if ( lookupSymbol != null)
35        {
36           iSymbols = coreSymbols + "," + lookupSymbol;
37        }

38        if ( iSymbols == null )
39        {
40           iSymbols = "";
41           System.out.println( "No symbols found." );
42        }
43        else
44        {
45           iStockQuotesList = stockQuoteService.getQuote( iSymbols );
46        }

47     }
48     catch ( Exception e )
49     {
50        e.printStackTrace();
51        throw new InitWidgetException( e.getMessage() );
52     }
53  }
```

12-14 Get the stock quote service using the `STOCK_QUOTE_SERVICE` property and store the name in the context for use by the `StockSymbolUpdate` pipeline component.

15 Get the mode the widget is in from the request context using a property name that is unique to the widget descriptor ID for the widget (necessary in case several different instances of the widget are on the same page). Note: managing the mode of the widget via this mechanism is a convention. Future releases of the framework will have more formal support for widget modes.

16-17 Set the widget mode to `DISPLAY` by default.

19 Pull the user ID from the session, where it has been stored by the simple login widget. Note: for widgets that use the Kinzan Common Authentication Framework, the mechanism to retrieve the common user ID is different.

20-27 If a user has logged in and has their user name already registered with the stock quote service, query the stock quote service for the list of stock symbols for the user. Otherwise, default to the default symbol list as defined by the `DEFAULT_SYMBOLS` property.

29-32 If the widget is not in `EDIT` mode, then the lookup symbol input field is active. Get the `LOOKUP_SYMBOL` form variable from the request context.

33-37 If the lookup symbol is not null, append it to the core symbols list so it gets displayed the next time the widget is rendered.

45 Query the stock quote service for the latest stock quotes for all the appropriate stock symbols.

The `getSymbols` method is an accessor to the `iSymbols` private variable (a comma-delimited list of stock symbols). The `getStockQuotesList` is an accessor method to the `iStockQuotesList` private variable (a linked list of `Quote` objects representing stock quotes for all the symbols in the symbols list).

To support the `PanelContentWidget` interface that allows the stock quote widget to be included in a panel widget, the `getTitle`, `isEditable`, `getEditWizard`, and `getEditState` methods are implemented as follows:

```
1  public String getTitle()
2  {
3     return (!"EDIT".equals( iWidgetMode ) ) ? "Stocks" : "Update Stocks";
4  }

5  public boolean isEditable()
6  {
7     return ( iUser != null && !"EDIT".equals( iWidgetMode ) );
8  }

9  public String getEditWizard()
10 {
11    return getProperty( "EDIT_WIZARD" );
12 }

13 public String getEditState()
14 {
15    return getProperty( "EDIT_STATE" ) + "?net_kinzan_requestContextID=" +
                 iContextID + "&WDID=" + iWd.getWidgetDescriptorId();
16 }
```

3 Return "Update Stocks" for the panel title if in `EDIT` mode, otherwise return "Stocks".

7 If the user is logged in and the stock quote widget is not already in `EDIT` mode, have the panel enable the edit button to enable switching the stock quote widget to `EDIT` mode.

11 Use the `EDIT_WIZARD` property of the stock quote widget as the wizard to be invoked by the panel widget when the edit button is pressed. This method is the default implementation from the `GenericPanelContentWidget`, and is included here for completeness.

15 Use the EDIT_STATE property of the stock quote widget, along with the request context ID and widget descriptor ID for the stock quote widget, as the state to be invoked by the panel widget with the edit button is pressed. In this case, the xSetDisplayMode action state will use the widget descriptor ID of the stock quote widget as a key to set the mode of the widget, and the request context ID is required by the State Manager.

4.4.3 Stock Quote Widget Template KCO File Implementation

The widget template KCO file uses the mode of the widget to determine the appropriate interface to present. Both DISPLAY and EDIT modes use the StockQuote widget instance to retrieve the appropriate stock and configuration information. DISPLAY mode has an input form that allows spot lookup of stock symbols, and EDIT mode has an input form that allows input of the preferred stock list for the user.[3]

The stockQuoteTemplate.kco file is as follows:

```
1  <%@ page import="net.kinzan.servlet.RequestContext"%>
2  <%@ page import="java.util.LinkedList"%>
3  <%@ page import="java.util.Iterator"%>
4  <%@ page import="com.kinzan.example.stockquote.widget.StockQuoteWidget"%>
5  <%@ page import="com.kinzan.example.stockquote.service.StockQuote"%>
6  <%RequestContext context =
           (RequestContext)request.getAttribute( "net.kinzan.requestContext" );
   %>
7  <%StockQuoteWidget myWidget = (StockQuoteWidget) stockQuote.getWidget(); %>
8  <%String stockListFormName = "stockList" + stockQuote.getWidgetDescriptorId();
   %>
9  <%String stockLookupFormName = "stockLookup" +
   stockQuote.getWidgetDescriptorId(); %>
10 <%String docRoot = context.getProperty( "site.documentRoot" ); %>

11 <%
12    String mode = context.getProperty( "net.kinzan." +
                                         stockQuote.getWidgetDescriptorId() +
                                         ".widgetMode" );

13    if ( session.getAttribute( "simpleLogin.user" ) == null || mode == null )
         mode = "DISPLAY";

14    if ( "EDIT".equals( mode ) )  // Begin EDIT mode
15    {
16  %>

17 <table width="100%" cellspacing="0" cellpadding="4" border="0">
18    <tr>
19       <td valign="middle">
20          <font>Use commas to separate stock symbols</font>
21       </td>
22    </tr>

23 <%
24    if ( context.getError() != null && context.getError().getMessage() != null )
25    {
26  %>
27    <tr>
```

[3] Note that widget mode support within widget template KCO files may be standardized and simplified in future releases. Specifically, widgets will be permitted to have multiple templates (each with their own variants) that are bound to the different widget modes. Collapsing both modes into the same widget template KCO is a workaround suitable for avoiding a need for this enhanced mode support.

```
28          <td valign="middle">
29             <font><b><%=context.getError().getMessage()%></b></font>
30          </td>
31       </tr>
32    <%
33       }
34     %>


35    <%
36       try
37       {
38          String symbolList = myWidget.getSymbols();
39          if (symbolList == null)
40             symbolList = "";
41     %>

42    <script>
43       function submitStockForm( state )
44       {
45          document.<%=stockListFormName%>.net_kinzan_nextState.value = state;
46          document.<%=stockListFormName%>.submit();
47       }
48    </script>

49       <tr>
50          <td valign="middle">
51             <form name="<%=stockListFormName%>" method="POST" action="/state">
52                <table cellspacing="0" cellpadding="3" border="0">
53                   <tr>
54                      <td valign="middle" align="left">
55                         <textarea name="NEW_SYMBOL_LIST" cols=20 rows=5>
56                            <%= symbolList %>
57                         </textarea>
58                      </td>
59                   </tr>
60                   <tr>
61                      <td valign="middle" colspan="2">
62                          <a href="javascript:submitStockForm(
      'xUpdateStockSymbolList' );"><img border="0"  '
      src="/<%=docRoot%>/resource/image/update_symbol.gif"></a> <a
      href="javascript:submitStockForm( 'xSetDisplayMode' );"><img border="0"
      src="/<%=docRoot%>/resource/image/back.gif"></a>
63                      </td>
64                   </tr>
65                </table>
66                <input type="HIDDEN" name="net_kinzan_nextState">
67                <input type="HIDDEN" name="net_kinzan_nextWizard"
                         value="StockQuoteWizard">
68                <input type="HIDDEN" name="net_kinzan_requestContextID"
                         value="<%=context.getID()%>">
69                <input type="HIDDEN" name="WDID"
                         value ="<%=stockQuote.getWidgetDescriptorId()%>">
70             </form>
71          </td>
72       </tr>

73    <%
74       }
75       catch (Exception e)
76       {
77          e.printStackTrace();
78       }
```

```
79  %>

80 </table>

81 <%
82    }    // End EDIT mode
83    else // Begin DISPLAY mode
84    {
85  %>

86 <table width="100%" cellspacing="0" cellpadding="1" border="0">
87    <tr>
88       <td>

89 <%
90       try
91       {
92          LinkedList newStockQuotesList = myWidget.getStockQuotesList();
93          if ( newStockQuotesList != null )
94          {
95  %>
96          <table width="100%" cellspacing="0" cellpadding="1" border="0">
97             <tr bgcolor="#cccccc">
98                <td width="33%" valign="middle" align="center">
99                   <font><b>Symbol</b></font>
99                </td>
101            <td width="33%" valign="middle" align="center">
102               <font><b>Price</b></font>
103            </td>
104            <td width="33%" valign="middle" align="center">
105               <font><b>Change</b></font>
105            </td>
106         </tr>
107<%
108         Iterator iterator = newStockQuotesList.iterator();

109         for ( ; iterator.hasNext() ; )
110         {
111            StockQuote quote = (StockQuote) iterator.next();

112            if (quote == null)
113               break;
114 %>
115            <tr>
116               <td width="33%" valign="middle" align="center">
117                  <font><%=quote.getSymbol()%></font>
118               </td>
119               <td width="33%" valign="middle" align="right">
120                  <font><%=quote.getPrice()%></font>
121               </td>
122               <td width="33%" valign="middle" align="right">
123                  <font><%=quote.getChange()%></font>
124               </td>
125            </tr>
126<%
127            } // for
128 %>
129         </table>
130<%
131         } // if newStockQuotesList != null
132      } // try
133      catch (Exception e)
134      {
```

```
135          e.printStackTrace();
136       }
137  %>

138       </td>
139    </tr>
140    <tr>
141       <td>
142          <form name="<%=stockLookupFormName%>" action="/state" method="POST">
143             <table width="100%" cellspacing="0" cellpadding="1" border="0">
144                <tr>
145                   <td valign="middle">
146                       <font><b>Get Quotes:</b></font>
147                   </td>
148                   <td align="right" valign="middle">
149                      <font><input type="text" name="LOOKUP_SYMBOL" size="10"
    maxlength="10"></font>
150                        <a
    href="javascript:document.<%=stockLookupFormName%>.submit();"><img border="0"
    src="/<%=docRoot%>/resource/image/go.gif"></a> 
151                   </td>
152                </tr>
153                <tr>
154                   <input type="HIDDEN" name="net_kinzan_requestContextID"
                          value="<%= context.getID()%>">
155                   <input type="HIDDEN" name="net_kinzan_nextWizard"
                          value="StockQuoteWizard">
156                   <input type="HIDDEN" name="net_kinzan_nextState"
    value="END">
157                </tr>
158             </table>
159          </form>
160       </td>
161    </tr>

162 </table>

163 <%
164    }   // End DISPLAY mode
165  %>
```

8-9 Generate unique form names for the stock list form and the stock look up form (in can multiple stock quote widgets get deployed to the same page).

12 Retrieve the widget mode from the context (note that this convention for mode management will be superceded by enhanced widget mode support in a future release).

13 Set the widget mode to `DISPLAY` by default.

14-82 The portion of the widget template KCO file that gets executed with the widget is in `EDIT` mode.

23-34 If a pipeline component has generated an exception (for instance, if the conditions of a validator are violated), output the error message on the page. The error message is retrieved from the context.

38 Query the widget for the list of stock symbols.

49-72 Data entry form in `EDIT` mode that allows the user to input a comma delimited list of stock symbols as their preferred stock symbols.

62 Links for two buttons: Update Symbols and Back. Update Symbols instructs the State Manager to execute the `xUpdateSymbolList` action state in the `StockQuoteWizard` to save the list of preferred stocks. Back instructs the State Manager to execute the `xSetDisplayMode` action state in the `StockQuoteWizard` to return the widget to `DISPLAY` mode.

66-69 Hidden form variables required by the State Manager and by the stock quote pipeline components.

83-164 The portion of the widget template KCO file that gets executed with the widget is in `DISPLAY` mode.

92 Retrieve the linked list of quote objects (each contains stock quote information for that particular stock), and if not null, display them.

97-106 Column headings for the stock quote information (three columns for symbol, price, and change)

107-114 For each quote object in the linked list...

115-125 ...output a table row with the stock quote information (symbol, price, and change).

4.4.4 Stock Quote Wizard Implementation

The stock quote wizard follows:

```
1  <?xml version="1.0"?>

2  <!DOCTYPE wizard SYSTEM "http://www.kinzan.net/dtd/wizard.dtd">

3  <wizard name="StockQuoteWizard">
4     <actionState name="xUpdateStockSymbolList"
         component="com.kinzan.example.stockquote.component.StockSymbolUpdate">
5        <transition event="SUCCESS" state="xSetDisplayMode"/>
6        <transition event="FAILURE" state="END"/>
7     </actionState>

8     <actionState name="xSetDisplayMode"
         component="com.kinzan.example.stockquote.component.SetDisplayMode">
9        <transition event="SUCCESS" state="END"/>
10       <transition event="FAILURE" state="END"/>
11    </actionState>

12    <actionState name="xSetEditMode"
         component="com.kinzan.example.stockquote.component.SetEditMode">
13       <transition event="SUCCESS" state="END"/>
14       <transition event="FAILURE" state="END"/>

15    </actionState>8    <actionState name="END"
         component="net.kinzan.state.pipeline.component.EndStateComponent"/>
16 </wizard>
```

3 Declare the name of the wizard.

4-7 Configure the `xUpdateStockSymbolList` action state (by convention, action states begin with "x") to use the `StockSymbolUpdate` pipeline component, and to transition to the `xSetDisplayMode` state when done so the widget can be returned to `DISPLAY` mode.

8-11 Configure the `xSetDisplayMode` action state, which uses a widget descriptor ID to change the mode of a stock quote widget to `DISPLAY` mode.

12-14 Configure the `xSetEditMode` action state, which uses a widget descriptor ID to change the mode of a stock quote widget to `EDIT` mode.

15 Configure the `END` state to use the provided `EndStateComponent`, which returns control to the state that initially invoked the wizard.

4.4.5 Stock Quote Widget Pipeline Components

The new methods in `StockSymbolUpdate` class are as follows:

```
1  public StockSymbolUpdate()
2  {
3     super();

4     ComponentParameter symbol = new ComponentParameter( "NEW_SYMBOL" );
5     symbol.setNullOkay( false );
6     iInputParameters.addComponentParameter( symbol );
7  }

8  public String processEvent( RequestContext context )
      throws InvalidParameterException, TransactionAbortedException,
             TimedOutException
9  {
10    super.processEvent( context );

11    String result = FAILURE;

12    String userID =
         (String) context.getSession().getAttribute( "simpleLogin.user" );
13    String symbolList = context.getProperty( "NEW_SYMBOL_LIST" );

14    try
15    {
16       StockQuoteService quoteService = (StockQuoteService) context.getService(
   context.getProperty( "net.kinzan.stockQuoteServiceName" ) );

17       if ( quoteService == null )
            throw new TransactionAbortedException( "Error! quote service not
   found!" );

18       // Use the service to store the stock symbols for the user.
19       if (quoteService != null)
20       {
21          quoteService.setSymbolByUser( userID, symbolList );
22          result = SUCCESS;
23       }
24    }
25    catch ( RemoteException e )
26    {
27       throw new TransactionAbortedException( e.getMessage() );
28    }

29    return result;
30 }
```

4 Construct a validator for the `NEW_SYMBOL_LIST` form variable (passed to the pipeline component by the widget).

5 Configure the validator to throw an exception if the parameter is null.

6 Add the parameter to the list of parameters to be automatically validated when the pipeline component is invoked.

12 Pull the user ID from the session, where it has been stored by the simple login widget. Note: for widgets that use the Kinzan Common Authentication Framework, the mechanism to retrieve the common user ID would be different.

13 Get the `NEW_SYMBOL_LIST` form variable from the request context.

16 Retrieve the stock quote service name from the context and establish a connection with the stock quote service.

21 Invoke the stock quote service to save the preferred symbols for the user. If the user does not exist in the stock quote service, the stock quote service automatically registers the user and stores their preferred stock list.

22 Set the return event for the pipeline component to `SUCCESS`, which will be used by the State Manager to invoke the next state in the wizard (in this case, the `xSetDisplayMode` action state, which will return the stock quote widget to `DISPLAY` mode).

The new methods in `SetEditMode` class are as follows:

```
1  public class SetEditMode
       extends BasicComponent
2  {

3      public SetEditMode()
4      {
5          super();

6          ComponentParameter wdID = new ComponentParameter( "WDID" );
7          wdID.setNullOkay( false );
8          iInputParameters.addComponentParameter( wdID );
9      }

10     public String processEvent( RequestContext context )
           throws InvalidParameterException, TransactionAbortedException,
                  TimedOutException
11     {
12         super.processEvent( context );

13         String wdID = context.getProperty( "WDID" );
14         context.setProperty( "net.kinzan." + wdID + ".widgetMode", "EDIT" );

15         return SUCCESS;
16     }

17 }
```

6 Construct a validator for the `WDID` form variable (passed to the pipeline component by the widget).

7 Configure the validator to throw an exception if the parameter is null.

8 Add the parameter to the list of parameters to be automatically validated when the pipeline component is invoked.

13 Get the `WDID` form variable from the request context.

14 Update the widget mode to `EDIT` in the context, using the widget descriptor ID as the key. Note: managing the mode of the widget via this mechanism is a convention. Future releases of the framework will have more formal support for widget modes.

The new methods in `SetDisplayMode` class are as follows:

```
1  public class SetDisplayMode
      extends BasicComponent
2  {

3     public SetDisplayMode()
4     {
5        super();

6        ComponentParameter wdID = new ComponentParameter( "WDID" );
7        wdID.setNullOkay( false );
8        iInputParameters.addComponentParameter( wdID );
9     }

10    public String processEvent( RequestContext context )
         throws InvalidParameterException, TransactionAbortedException,
                TimedOutException
11    {
12       super.processEvent( context );

13       String wdID = context.getProperty( "WDID" );
14       context.setProperty( "net.kinzan." + wdID + ".widgetMode", "DISPLAY" );

15       return SUCCESS;
16   }

17 }
```

14 Essentially the same as the `SetEditMode` class. Update the widget mode to `DISPLAY` in the context, using the widget descriptor ID as the key. Note: managing the mode of the widget via this mechanism is a convention. Future releases of the framework will have more formal support for widget modes.

4.4.6 Sample Page

This sample shows how the simple weather widget may be used on a page:

```
1  <widget type="simplePanel" binding="content">
2     <widgetIncludeList>
3        <widget type="stockQuote" binding="content"/>
4      </widgetIncludeList>
5  </widget>
```

1 Because the stock quote widget has been designed to work with a panel widget, it is necessary to instantiate a panel widget that has the stock quote widget as an included widget. In this case, `simplePanel` uses a `GenericPanelContentWidget` for its widget class (defined in the widget factory declaration for `simplePanel` in the kapp file). The widget template KCO for the `simplePanel` widget is described in the next section.

2-4 Include the stock quote widget with the binding name of "`content`". This is used by the `simplePanel` widget to substitute the stock quote widget into its content area. It is also used by the `simplePanel` widget to query the stock quote widget to configure the title and edit button on the `simplePanel` widget.

4.4.7 Simple Panel Widget Template KCO

The simple panel widget coordinates with the stock quote widget to manage the display. The coordination is via the `PanelContentWidget` interface (which the stock quote widget must implement) and via the widget template KCO for the simple panel widget.

By using common user interface widgets in an application, the user interface for an application may be very rapidly modified or changed to reflect new branding or user interface standards. Widgets also are made more generic, allowing them to be used with may different user interfaces.

The `simplePanelTemplate.kco` file is also a good example of how nested widgets may communicate with one another. The widget template KCO is as follows:

```
<%@ page import="net.kinzan.servlet.RequestContext"%>
<%@ page import="net.kinzan.rendering.PanelContentWidget"%>
<%RequestContext context =
    (RequestContext)request.getAttribute( "net.kinzan.requestContext" ); %>

<widgetInclude binding="content" action="import"/>

<%
    PanelContentWidget contentWidget = (PanelContentWidget)content.getWidget();
    String docRoot = context.getProperty( "site.documentRoot" );
    String editLink = "/state/" + context.getProperty( "site.documentRoot" )
                      + "/" + contentWidget.getEditWizard()
                      + "/" + contentWidget.getEditState();

    String width = contentWidget.getProperty( "WIDTH" );
    if( width == null )
       width = "100%";
 %>

<table width="<%=width%>" cellspacing="0" cellpadding="1" border="0">
   <tr>
      <td>
        <table width="100%" cellspacing="0" cellpadding="1" border="0"
    bgcolor="black">
            <tr>
               <td valign="middle">
                  <font color="#ccff00"><b><%=
  contentWidget.getTitle()%></b></font>
               </td>
<%
   if ( contentWidget.isEditable() )
   {
 %>
               <td align="right" valign="middle">
                   <a href="<%=editLink%>"><img border="0"
  src="/<%=docRoot%>/resource/image/edit.gif"></a>
               </td>
<%
   }
 %>
            </tr>
         </table>
      </td>
   </tr>

   <tr>
      <td>
         <widgetInclude binding="content" action="display"/>
      </td>
   </tr>
```

```
38 </table>
```

---

3 Import the included widget with binding name `"content"`. A widget proxy for the included widget with variable name `content` is automatically available within the KCO.

5 Use the widget proxy to get the included widget and cast it to a `PanelContentWidget` interface. (Recall that the panel widget requires that any widget that it includes to implement this interface).

7 Construct the URL that the edit button will link to. The wizard and state are retrieved by the included content widget (the `getEditWizard` and `getEditState` methods are part of the `PanelContentWidget` interface).

8-10 Get the preferred width for the included content widget (in this case, a widget property called `WIDTH`).

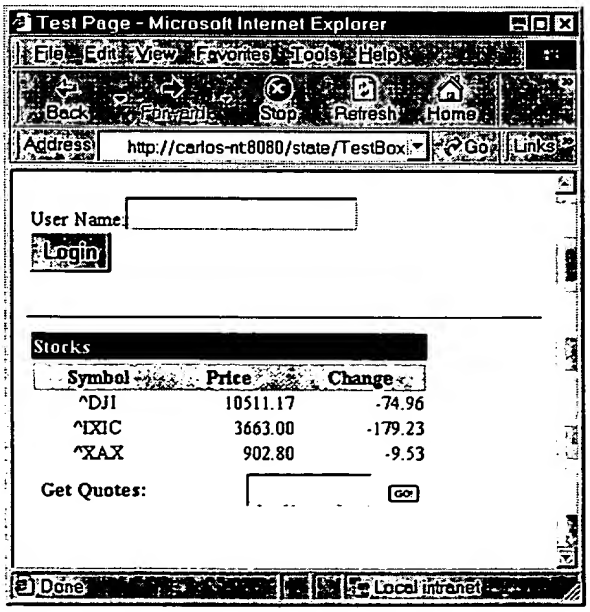
13-32 Construct the top of the panel, messaging the included content widget for the title to display on the panel (the `getTitle` method) and displaying the edit button if the included content widget responds that it is editable (the `isEditable` method). The action for the edit button was constructed on line 7.

24 Note the use of the document root to construct the URL for an image resource that is in the resource directory for the application (recall that all items in the `/resource` directory are automatically deployed to the document root for the application without processing). This is the preferred technique for referencing static resources that do not require variants based on style, locale, or target device.

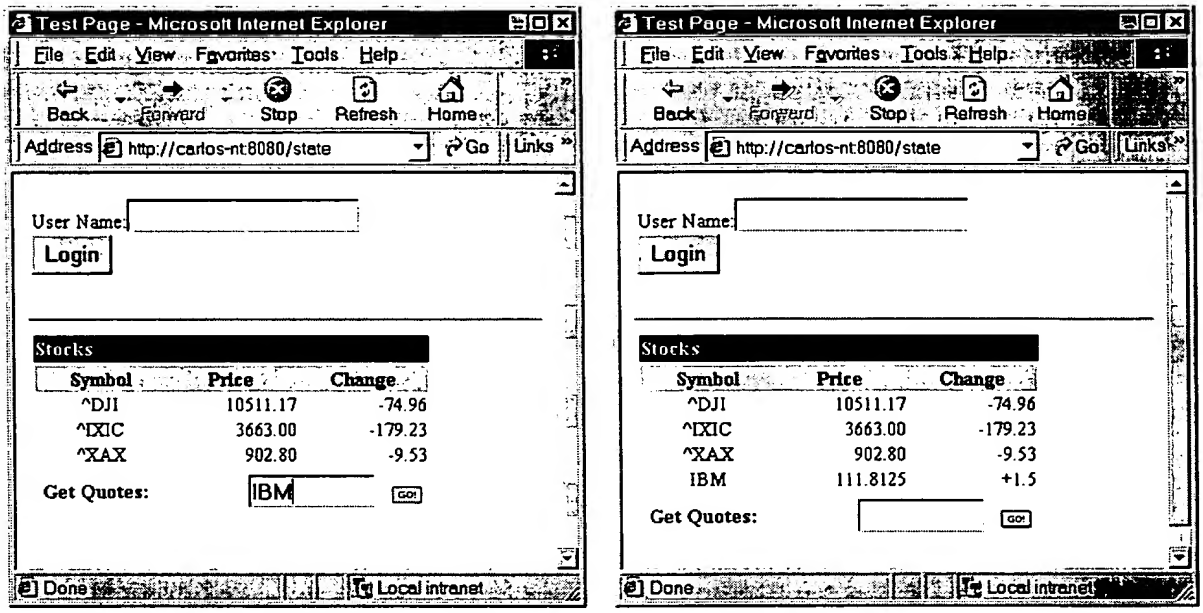
33-37 Construct the content area of the panel.

35 Display the included widget with binding name of `"content"`.

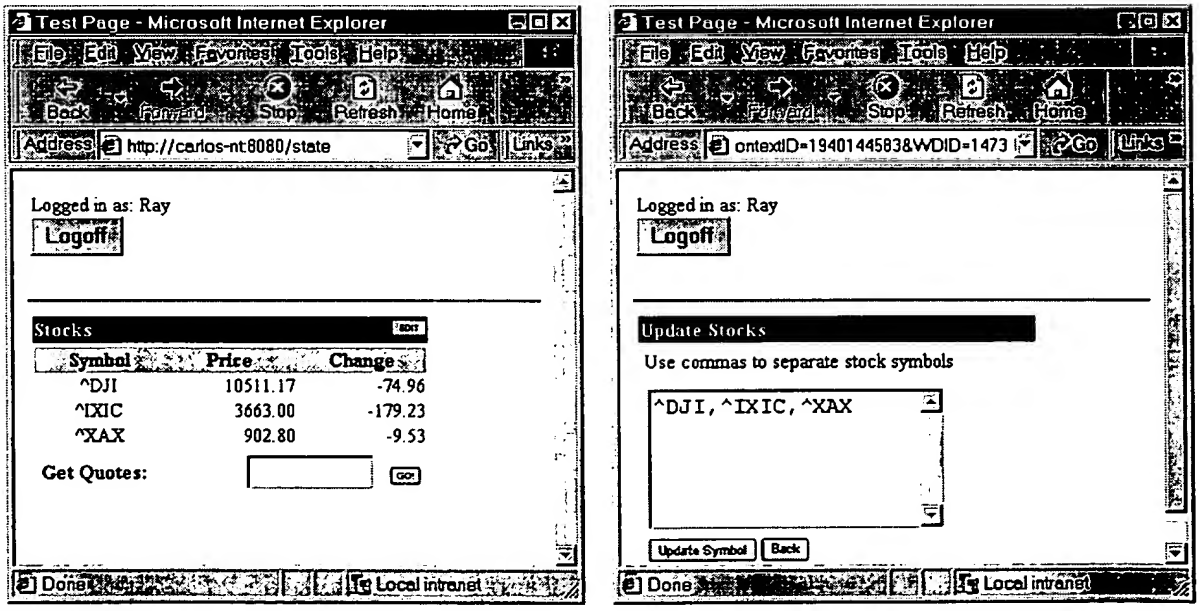
The result of both the simple panel widget and the stock quote widget is as follows. (Note that these screen shots also include a simple login widget at the top of the page. Simple login was necessary to allow full testing of the stock widget).



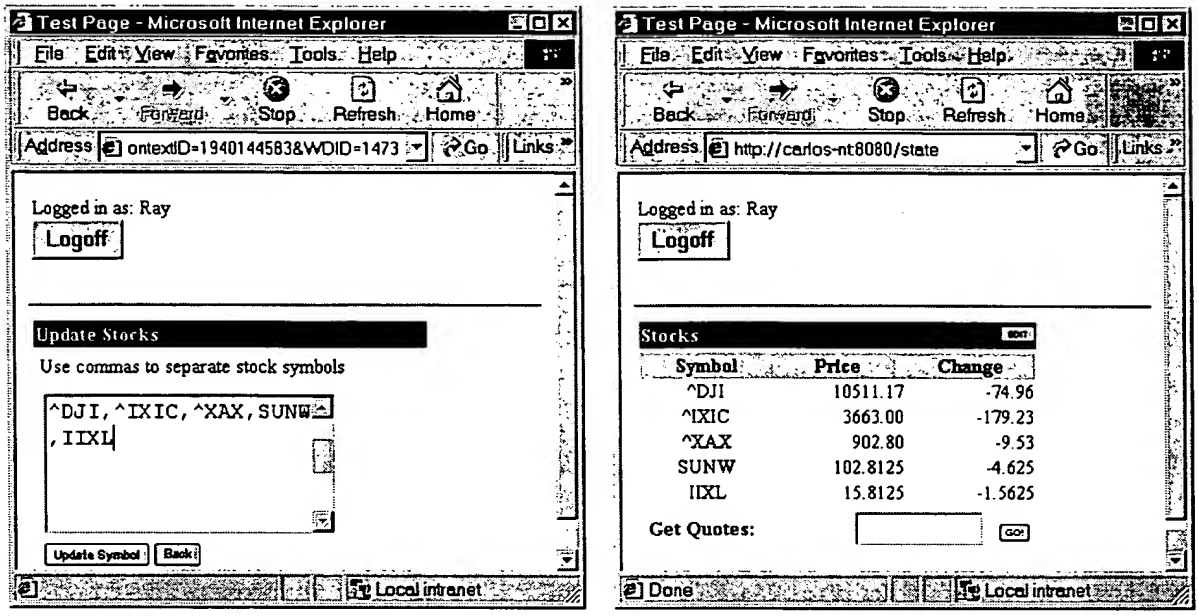
Spot look up of the stock quote for IBM appends the quote to the end of the core group of stock symbols:



Once logged in, the simple panel widget displays the edit button in the upper right corner, which allows the user to switch the widget to `EDIT` mode (in place editing of the widget). Once in `EDIT` mode, the edit button is no longer available, and the title displayed on the panel has changed.



The user may then input the stock symbols they are interested in, which returns the widget to `DISPLAY` mode. The users stock preferences are managed via personalization EJBs by the stock quote service, are thus shared across all applications that use the stock quote service.



5 Widget API

The following section describes one embodiment of a widget API in an exemplary manner.

5.1 Standard Properties

In one embodiment, the factory or the widget may define properties. Properties defined by a factory are accessible to all widgets created by that factory. Properties defined by a widget take precedence over the factory widget properties. The Rendering Service at runtime uses the `JSFILE` property if defined. This property must contain the full path to the JavaScript file that should be included in the header portion of the HTML page. The JavaScript file should wrap its contents with the `<script>` tag.

5.2 Java Classes

This section provides a brief description of the Java classes that form part of the widget API in one embodiment. For a more detailed description of these classes please see the Java documentation.

`GenericWidget`

The `GenericWidget` class provides a default implementation of the `Widget` interface. Widget developers are strongly encouraged to extend this class when developing new widgets. Please see the `Widget` interface for additional information.

`GenericWidgetFactory`

This class provides a default implementation of the `WidgetFactory`. Widget developers are strongly encouraged to extend this class when developing new widgets. This class returns a `GenericWidget` when invoked by the Rendering Service. Please see the `WidgetFactory` for additional information.

`WidgetFactory`

This class declares an interface that all widget factories must implement. The Rendering Service uses this interface to initialize a new widget factory and instantiate new widgets. Widget developers would

seldom have to work with this interface since a standard implementation is provided in the `GenericWidgetFactory` class.

`void init (AppContext appContext, WidgetFactoryDescriptor factoryInfo)`
The Rendering Service calls this method once to initialize the factory. One-time configuration operations should be implemented in this method. The Rendering Service passes a `WidgetFactoryDescriptor` object containing the widget factory properties defined by the user.

`Widget createWidget( RequestContext context, WidgetDescriptor wd )`
The Rendering Service calls this method every time a new widget needs to be created. The `RequestContext` is part of the Kinzan Servlet Framework. It contains the HTTP servlet request and reply. The widget factory may use the information contained in the request context to create a specific type of widget. The `WidgetDescriptor` class contains the widget properties.

`String getProperty( String propertyName )`
This method provides access to the widget factory property with the specified name.

`String[] getProperties( String propertyName )`
This method provides access to all widget factory properties that have the specified name.

`WidgetProxy`
The `WidgetProxy` interface defines a set of methods that a widget template KCO file may use to style and layout the widget on the page. The Rendering Service uses the `WidgetProxy` to decouple page-specific information from the widget.

`Style getStyle()`
This method returns the style object that should be applied to the widget. The KSP compiler uses this `Style` object to style the JSP page.

`String getName()`
This method returns the name that is used to identify the widget outside its JSP file.

`Integer getOrdinal()`
This method returns the ordinal used by the JSP page to order the widget on the page.

`Widget getWidget()`
This method returns the actual widget object.

`Integer getWidgetDescriptorId()`
This method returns the ID used to identify the widget descriptor. This widget descriptor was used to create the Widget contained in the proxy.

`Page getPage()`
This method returns the `Page` object that contains the current widget.

`WidgetProxy getChildByName( String name )`
This method returns a child widget proxy with the name specified. The method returns null if the widget does not have any child widgets with the specified name.

`WidgetProxy[] getChildren()`
This method returns an array of all children widgets. It the current widget is not a container, the method returns null.

`Widget`
All widget instances must implement this interface. The widget presentation servlet and the Rendering Service use this interface to get access to the widget properties.

`String getName()`
This method gets the unique identifier for a widget. This identifier is unique within a site.

`String getParameterName()`
This method gets the parameter name that should be used to identify this widget and its widget proxy within its corresponding KCO page. The parameter name is the name attribute of the widget factory.

`WidgetFactory getWidgetFactory()`
Gets the widget factory that was used to create this widget.

`String getProperty( String propertyName )`
This method returns the widget property specified by the `propertyName` parameter.

`String[] getProperties( String propertyName )`
Get all the properties that have the specified property name.

```
void init( RequestContext reqCtx, WidgetDescriptor wd,
           WidgetFactory factory )
```
This method is called once by the corresponding widget factory to initialize the widget. The factory passes the request context, the widget descriptor and a reference to itself. The request context can be used to query the servlet session, request and reply. The `WidgetDescriptor` contains the properties for the widget that is being initialized.

6 Java and XSL Examples

The following section provides examples of embodiments illustrated in Java and XSL.

6.1 SimpleSyndicatedContentWidgetFactory.java

```
/*
 * This class was implemented for demonstration purposes. It should not be
 * used for production. Error checking has not been implemented to make the
 * code easier to read.
 */

package com.kinzan.example.simplesyndicatedcontent.widget;


import java.net.URL;
import java.io.InputStream;

import net.kinzan.thread.TimeValue;
import net.kinzan.thread.TimeoutException;
import net.kinzan.util.AgingCachePolicy;
import net.kinzan.util.WeakCache;

import net.kinzan.servlet.RequestContext;
import net.kinzan.servlet.AppContext;
import net.kinzan.rendering.Widget;
import net.kinzan.rendering.InitWidgetFactoryException;
import net.kinzan.rendering.InitWidgetException;
import net.kinzan.rendering.CreateWidgetException;
import net.kinzan.rendering.model.WidgetDescriptor;
import net.kinzan.rendering.model.WidgetFactoryDescriptor;
import net.kinzan.rendering.component.GenericWidgetFactory;

public class SimpleSyndicatedContentWidgetFactory extends GenericWidgetFactory
{
   private WeakCache iContentCache = null;
```

```
/**
 * Initializes the Syndicated Content widget factory.  It gets the
 * AGING_INTERVAL property and creates the Content cache.
 * @param  factoryInfo widget factory configuration
 * @return none
 * @exception InitWidgetFactoryException if there is an error in the super class
 * init.
 */
public void init( AppContext appContext, WidgetFactoryDescriptor
factoryDescriptor )
   throws InitWidgetFactoryException
{
   super.init( appContext, factoryDescriptor );

   long maxCacheAge = Long.parseLong( getProperty( "AGING_INTERVAL" ) );

   iContentCache = new WeakCache( new AgingCachePolicy( maxCacheAge ) );

}

/**
 * Creates a Simple Syndicated Content widget.
 * It first checks if the content is in the cache.
 * If it is not in the cache, fetch the data.  This function then calls the
 * processXMLData in the widget.
 * @param context the RequestContext.  It not used in this widget.
 * @param wd The widget descriptor. This is passed to the the widget ctor.
 * @return A Simple Syndicated Content widget.
 * @exception CreateWidgetException if there is an error in creating the widget
 * or in getting the content from the URL.
 */

public Widget createWidget( RequestContext context, WidgetDescriptor wd )
   throws CreateWidgetException, InitWidgetException
{
   SimpleSyndicatedContentWidget sscWidget = null;
   String xmlData = null;

   try
   {
      if ( wd == null )
         throw new CreateWidgetException( "The WidgetDescriptor was null for
this request." );

      sscWidget = new SimpleSyndicatedContentWidget();
      sscWidget.init( context, wd, this );

      String xmlURL = sscWidget.getProperty( "URL" );
      if ( xmlURL == null )
         throw new CreateWidgetException( "The xml URL must be specified!" );

      xmlData = (String) iContentCache.get( xmlURL );

      if ( xmlData == null )
      {
         URL url = new URL( xmlURL );
         InputStream inStream = url.openStream();

         byte[] buffer = new byte[1024];
         StringBuffer tempString = new StringBuffer();

         int count = 0;
         while ( (count = inStream.read( buffer ) ) != -1)
```

```
            {
               tempString.append( new String( buffer, 0, count ) );
            }

            xmlData = tempString.toString();
            inStream.close();

            if ( xmlData != null )
            {
               iContentCache.put( xmlURL, xmlData );
            }
         }

         if ( xmlData != null )
            sscWidget.processXMLData( xmlData );
         else
            sscWidget.setContent( "Data not available!" );

      }
      catch ( Exception e )
      {
         throw new CreateWidgetException( e.getMessage() );
      }

      return sscWidget;
   }

}
```

6.2 SimpleSyndicatedContentWidget.java

```
/*
 * This class was implemented for demonstration purposes. It should not be
 * used for production. Error checking has not been implemented to make the
 * code easier to read.
 */

package com.kinzan.example.simplesyndicatedcontent.widget;

import java.io.IOException;
import java.io.FileReader;
import java.io.StringReader;
import java.io.StringWriter;
import java.net.MalformedURLException;

import org.xml.sax.SAXException;
import org.apache.xalan.xslt.XSLTProcessor;
import org.apache.xalan.xslt.XSLTInputSource;
import org.apache.xalan.xslt.XSLTProcessorFactory;
import org.apache.xalan.xslt.XSLTResultTarget;

import net.kinzan.servlet.RequestContext;
import net.kinzan.rendering.Widget;
import net.kinzan.rendering.WidgetFactory;
import net.kinzan.rendering.InitWidgetException;
import net.kinzan.rendering.model.WidgetDescriptor;
import net.kinzan.rendering.component.GenericWidget;
import net.kinzan.rendering.component.GenericWidgetFactory;

public class SimpleSyndicatedContentWidget extends GenericWidget
{

   private String iContent;
```

```
   /**
    * Gets the HTML fragment created by the processXMLData method.
    * @param none
    * @return The HTML fragment.
    * @exception none
    */

   public String getContent()
   {
      return iContent;
   }

   public void setContent( String newContent )
   {
      iContent = newContent;
   }

   /**
    * Processes the XML data into a HTML fragment using XSL transformation.  This
    * method is invoked by the SSC widget factory after this widget is created.
    * @param  xmlData  the XML data.
    * @return none.
    * @exception IOException if there is an error in reading the XSL file.
    * @exception SAXExcpetion if the XSL processor encounters an error.
    */

    public void processXMLData( String xmlData )
       throws IOException, SAXException
   {

      FileReader xslFile = new FileReader( getProperty( "XSL" ) );
      StringReader xmlSource = new StringReader( xmlData );

      StringWriter output = new StringWriter( 2048 );

      XSLTProcessor processor = XSLTProcessorFactory.getProcessor();

      processor.process( new XSLTInputSource( xmlSource ),
                         new XSLTInputSource( xslFile ),
                         new XSLTResultTarget( output ) );

      iContent = output.getBuffer().toString();

      xslFile.close();
   }

}
```

6.3 Simple Syndicated Content XSL File

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
   <xsl:output method="html"/>
   <xsl:template match="/moreovernews">
      <table>
      <xsl:for-each select="article">
         <tr>
         <td>
            <font face="helvetica" size="1">
            <a>
               <xsl:attribute name="href">
                  <xsl:value-of select="url"/>
```

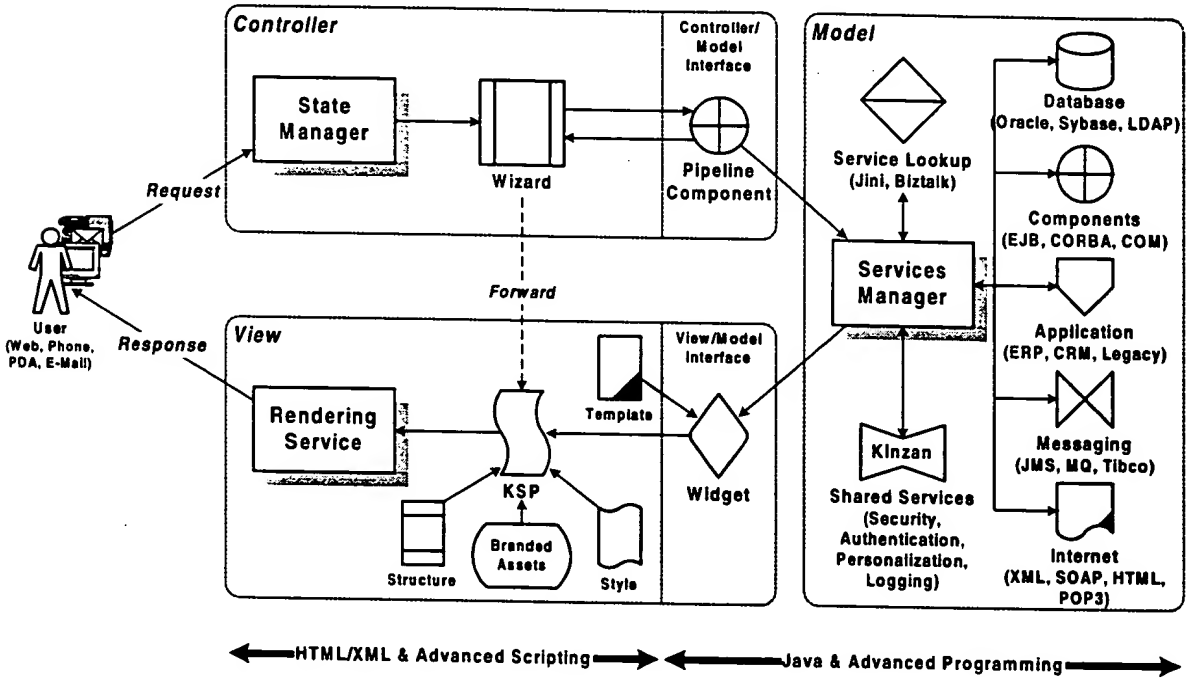
```
                </xsl:attribute>
                <xsl:value-of select="headline_text"/>
            </a>
            </font>
            <font size="1" face="san-sheriff">
                <br/> <i>(<xsl:value-of select="source"/>,
                        <xsl:value-of select="harvest_time"/> )</i>
            </font>
        </td>
        </tr>
    </xsl:for-each>
    </table>
  </xsl:template>
</xsl:stylesheet>
```

Kinzan State Manager Developer's Guide

1 Introduction

This document provides an overall view of how pages and application logic may be linked into a web application using an embodiment of the Kinzan State Manager. It also describes the API that must be used in that embodiment to create and customize application logic components (referred to as pipeline components) and gives some examples (including example code).

2 The Big Picture



As discussed in the "KTP Framework Overview" document, in one embodiment, wizards, pipeline components, and the Kinzan State Manager represent the "Controller" in the Kinzan Technology Platform (KTP) architecture.

In contrast to the use of monolithic controller servlets in a typical Java Model 2 architecture, wizards allow the easy modularization and dynamic assembly of application login and transitions between different states of an application.

In one embodiment of the KTP architecture, interfacing between the controller and model layers (typically a very daunting component of a multi-tier architecture for non-programmers) is consolidated into easily reusable pipeline components. Often, widgets are closely associated with their own wizards, effectively preconfiguring the controller layer for web developers. This allows web developers to focus on creating compelling customer experiences and not the nitty gritty of multi-tier application development.

In some embodiments, all the various modular artifacts in the KSP framework can be inherited and shared between different sites and applications, encouraging true reuse and consistency when assembling (versus building) applications.

If the functionality of the preconfigured wizards needs to be adjusted, the various components of the wizard definition can be easily reordered, added to, or streamlined. The result is a highly modular and

configurable assembly approach to the controller layer, avoiding the development and maintenance overhead of unique controller servlets for each application.

The use of the Kinzan Rendering Service to apply a similar modular assembly approach to the view tier is described in the "KSP Developer's Guide".

This document introduces the basics of the State Manager development and runtime environments. Other developer's guides give more examples on how to integrate wizards into applications.

3 Kinzan State Manager Overview

The Kinzan State Manager is a very powerful and flexible tool for assembling web applications. Using various embodiments of the State Manager, developers can rapidly assemble web pages (typically Kinzan Server Pages (KSPs)) and application logic to configure their application.

The State Manager supports flexible application configuration by cleanly separating state management from the presentation layer, and providing a mechanism to chain application logic modules to connect application pages. It also provides for management of state at the user and request level, avoiding many concurrency issues that plague less sophisticated state management systems (including duplicate request handling).

Application logic is implemented in modules using Java objects called pipeline components. These modules allow for "plugable" components that may be assembled dynamically based on user input and the current state. Components may be developed at a more atomic level and chained together, giving more flexibility in modifying or customizing applications for different users. The State Manager also provides for convenient and extremely powerful field validation by these components.

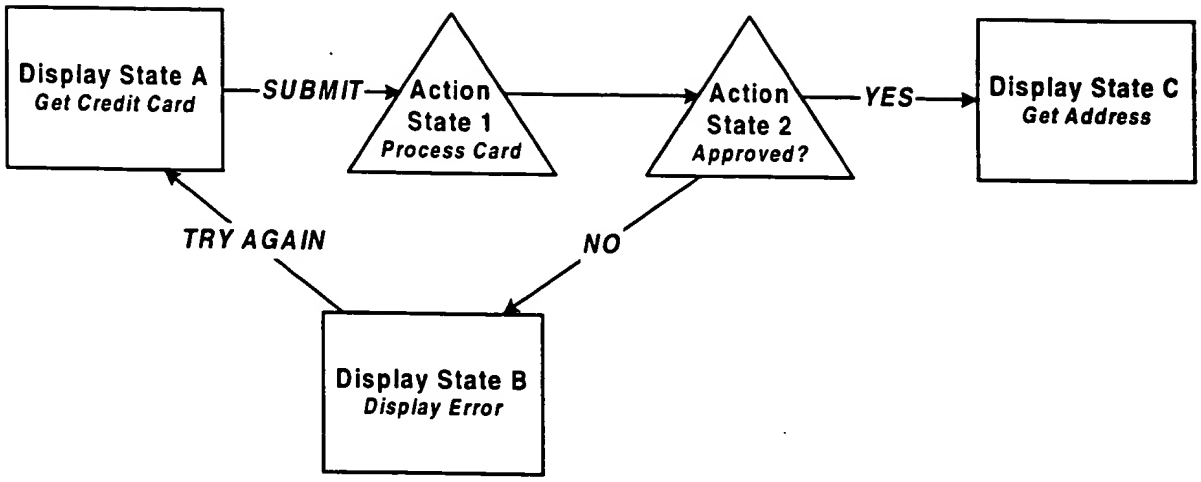
The State Manager provides basic transaction management across pipeline components, and facilitates linkages to business logic (usually Enterprise Java Beans (EJBs)) and various support services.

Applications may be defined to have many mini-applications (or wizards) within them, allowing better logical segmentation of application functions. This segmentation encourages reuse of wizards within and between applications, streamlining application development. The State Manager supports coordination and handoff between wizards.

Wizards are defined using XML, allowing for flexible configuration of wizards by less sophisticated developers, with easy integration into support tools.

3.1 What Is A State Diagram?

A state diagram (interchangeably referred to as a "wizard" in this document) represents discrete steps in an application flow, with decision points that trigger transitions to different sections of the application.



For example, in the figure above, the application begins in "Display State A". When the user enters their credit card number and generates a "SUBMIT" event by pressing submit on a form, the application moves to "Action State 1". "Action State 1" processes the credit card number and forwards the results to "Action State 2". "Action State 2" applies business rules to the results of the credit card processing to determine whether the merchant will accept the credit card.

If the rules in "Action State 2" generate a "YES" event, control is passed to "Display State C" which presents the user with a form to enter their address. If the rules in "Action State 2" generate a "NO" event, control is passed to "Display State B" which presents the user with a rejection message and gives them the opportunity to enter another credit card number.

In "Display State B", if the user chooses to enter another credit card number (perhaps by pressing a "Try Again" link), a "TRY AGAIN" event is generated and control is passed back to "Display State A" for the process to begin again.

3.2 Advantages Of Using A State Diagram

HTTP is a stateless protocol. Once a web server transfers a web page to a web browser, it has no knowledge about how pages are related to one another. This is one reason why links are embedded within web pages instead of being managed by the server.

By utilizing a state manager (with appropriate state diagrams), application developers can describe sophisticated relationships within an application, without being forced to manage all sorts of tricks involving manipulation of the actual web pages.

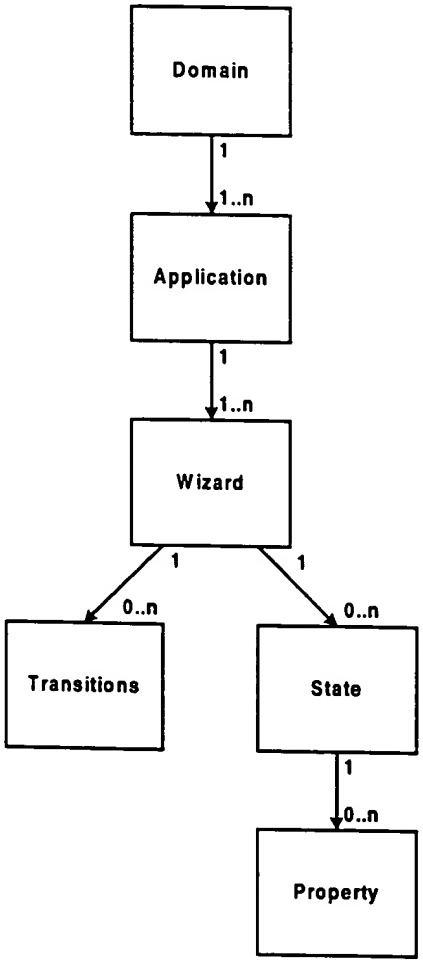
Once developers begin to model their applications using traditional state diagrams, they immediately benefit from enhanced documentation, manageability, flexibility, and reuse across applications.

For instance, in the example in the previous section, a developer could easily replace "Action State 2" with a different piece of business logic that changes the criteria used to approve or deny a credit card transaction. They could do so by replacing a single component, rather than having to dig through a much larger piece of application logic that was distributed through lots of presentation code.

If, in the future, the application needs to be more sophisticated to support multiple approval components for different merchants, the modification to the application would be straightforward. The developer could insert a piece of application logic between "Action State 1" and "Action State 2" to determine which of many possible approval components should be invoked for a particular merchant and trigger that approval component, all without impacting the other components of the application (logic or presentation).

3.3 Components Of The Kinzan State Manager

The following figure represents the various components of the State Manager model in one embodiment:



3.3.1 Domain

A domain is a collection of applications. It is the interface through which all state requests should be made in order to support inheritance of applications, wizards, etc. between parent and child sites.

3.3.2 Application

An Application is a collection of state diagrams that are associated with a particular application. By grouping many wizards in an Application, there is a common deployment group for a particular application. Applications may be defined as children of other Applications. A child application inherits Wizards, States, and Transitions from its parent application.

3.3.3 Wizard

A Wizard (or state diagram) is a set of states, events, and pointers to pipeline components and pages. An example would be a Tax Wizard or a Shipping Wizard.

3.3.4 State

A state is a location in the state diagram being managed by the State Manager.

States may be either display states (which reference a KSP or URL) or action states (which ref renc a pipeline component for processing)

3.3.5 Transitions

States may have zero to many transitions to other states, which are triggered by various events.

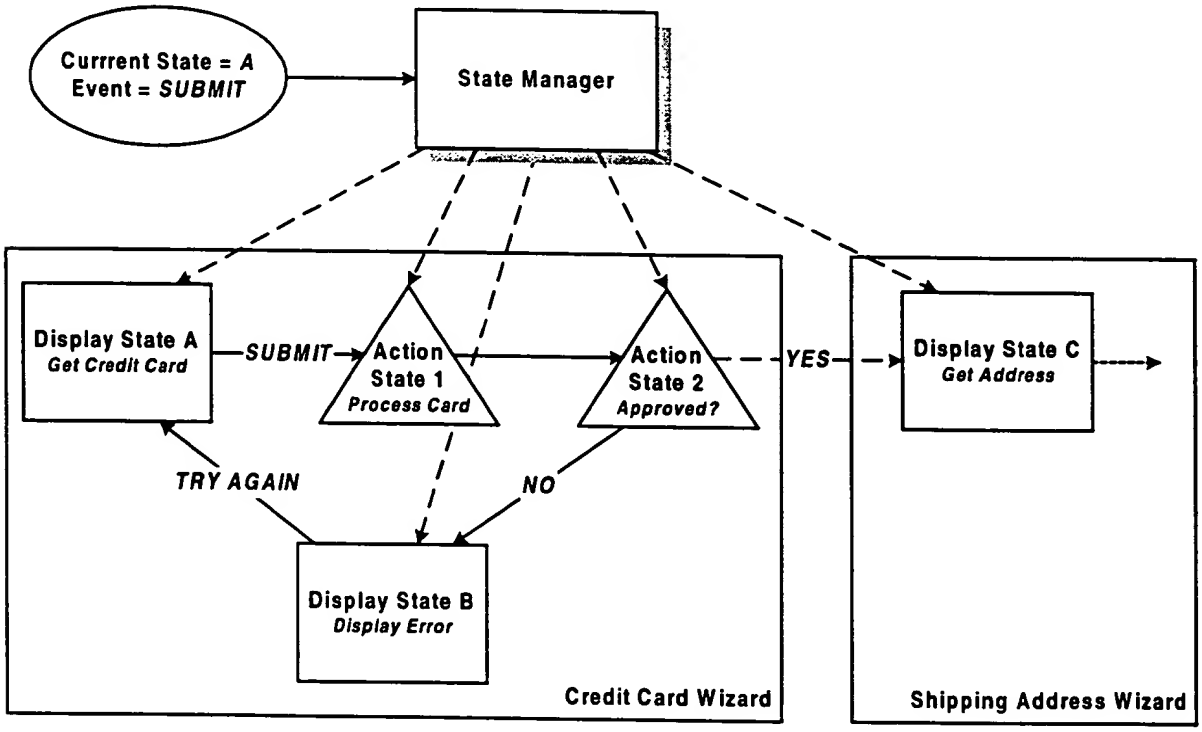
3.3.6 Properties

State may also have properties associated with them as name-value pairs. This may be useful in configuring a shared piece of application logic (or Action State) for a particular application.

4 State Manager Details

4.1 State Manager

In one embodiment, the State Manager controls execution of a Domain. Based on a current state and an input event, it retrieves the next `State` object. If the state is an action state, the associated code is executed using the current `RequestContext` as input. The State Manager processes the resulting event and retrieves the next `State` object. This loop continues until a display state is triggered, at which point control is returned to the calling code.



For example, in the above figure the State Manager is invoked as the "ACTION" from a web page, which passes some form variables with a "SUBMIT" event. Because the application is in "Display State A", the State Manager queries the transitions for "Display State A" to determine that control should be passed to "Action State 1" on a "SUBMIT" event. "Action State 1" then retrieves the form variables from the request context and executes its application logic.

The State Manager manages all wizards in an application. In the above example, a "YES" event from "Action State 2" passes control to the entry point of the "Shipping Address Wizard".

4.2 State Types

States may be associated with a display page or with an action.

If the state is associated with a display page, the page will be returned to the user and the State Manager will wait for input. One of the outputs from this page will be an event, which will be passed to the State Manager to determine the next state.

If the state is associated with an action, the action will be executed when the state is entered. The code will return an event, which the State Manager will use to determine the next state.

`DisplayState`
The `DisplayState` class represents a state with an associated user-interaction page. The user-interaction page is referred to as either a KSP (using the `ksp` attribute) or as a redirect URL (using the `forward` attribute). Only display states are stored in a user's state history.

`ActionState`
The `ActionState` class represents a state with an associated action. It contains a reference to a pipeline component, which is the interface for defining actions.

`ActionStates` have two default events: "`SUCCESS`" and "`FAILURE`". They must have a `SUCCESS` event defined. A `FAILURE` state is optional because Wizards and Domains have default failure states.

An `ActionState` may also be used to take input and make a decision about the next appropriate output state. A simple example would be an `ActionState` whose output event is the reseller ID of the user. The application developer could then use the reseller ID as an event to trigger the next state based on the ID of that reseller.

4.3 Pipeline Component

The pipeline component interface defines a single piece of code that is associated with a State. Its main method receives a context (which includes session and request information), performs an action, and returns an output event. This output event leads to another State, which may be either another `ActionState` or a `DisplayState`.

4.4 State Servlet

The State servlet manages retrieval of the parameters from the form. It places the parameters into an object and passes them to the State Manager for processing. When the call to the State Manager returns, the State servlet dispatches a request to the resulting KSP file, or redirects to a new URL, depending on what was returned by the State Manager.

4.5 Typical Sequence Of Events

In a typical use of the State Manager in one embodiment, the State servlet is invoked as the action from an HTML form. It gets passed the various form parameters and session information, packages them, and passes them to the State Manager.

The State Manager retrieves the request. It expects to see parameters named `net_kinzan_nextWizard`, `net_kinzan_nextState`, and `net_kinzan_nextEvent`. It determines the next state by using the appropriate `Domain`, `Wizard`, and `State` classes (even if different that the current wizard and current state).

Alternatively, if the `net_kinzan_requestContextID` parameter is set to the request context ID of the form, the `net_kinzan_nextWizard` and `net_kinzan_nextState` parameters are assumed

to be the same as the current wizard and current state. The `net_kinzan_nextEvent` parameter is still used to determine the next event.

If the next state is an `ActionState`, the state's pipeline component is retrieved and executed. The resulting event is then used to retrieve the next state and which is then similarly processed.

If the next state is a `DisplayState`, control is returned to the caller, with the `DisplayState` as the return value. The State servlet then dispatches control to the appropriate KSP file (the display state defines a `ksp` attribute), or redirects to a new URL (if the display state defines a `forward` attribute).

4.6 Wizards As Independent Mini-Applications

One of the major design abstractions when using the State Manager is the wizard. A wizard is a set of states that act as an independent mini-application. Encapsulating these mini-applications encourages reuse within and between applications. For example, a Tax Wizard for configuring tax rate may be accessible from either a store-install Wizard or from a central administration page.

While decomposing an application into many mini-applications is useful and powerful, it does create the need to coordinate control between wizards. For example, if a Tax Wizard is used in multiple areas of a web site, how does the Tax Wizard know where to return when it finishes? The Tax Wizard should not have to know what other states or pages might link to it or have to hardcode the return state, since this would limit many of the advantages from encapsulation.

The State Manager supports coordination between wizards by maintaining a call stack that tracks "START" and "END" states between wizards. When a wizard reaches its own "END" state, the State Manager returns control to the state that previously had called the wizard. Alternatively, the state that called the wizard may give the State Manager an explicit return location. To use this feature so as to not hardcode a wizard's return state, make a final state named "END", linked to a component named `EndStateComponent`.

The State Manager determines the return location for a wizard in one of two ways. If a request to that Wizard is made with the parameter "RETURN_URL", the value of this parameter becomes the return location of the wizard. If no such parameter exists, the state prior to entering the new wizard is set as the return location of the wizard.

In either case, when the END state is reached, the return location is retrieved by the `EndStateComponent`, which directs the State Manager to move to that location in the state diagram.

Below is a list of classes used to implement this feature, along with descriptions.

`EndStateComponent`
The `EndStateComponent` is a `PipelineComponent` that redirects the application to the wizard's return location. To do this, it checks for a return location in the `ReturnStack`. If it finds one, it throws a `JumpToStateException` constructed with the return location. The `JumpToStateException` is then caught by the State Manager, which gets the return location from the `JumpToStateException` and directs the application to that state.

`ReturnStack`
The `ReturnStack` encapsulates a stack of return locations for an application session. It is important to have a stack (rather than a single variable) because Wizards may call other Wizards, and each has a different return location. Without a stack the return location would be overridden which each new transition.

`JumpToStateException`
A `JumpToStateException` directs the State Manager to move to a new state, circumventing the event driven state transition model. A `PipelineComponent` may throw this exception whenever it needs to move to a state that cannot be reached by any transition.

4.7 Miscellaneous Solution D tails

The following details are exemplary in nature, relating to specific embodiments, and may not be necessary in all embodiments.

4.7.1 History

The State Manager may maintain a history for each wizard. This history maintains session and page information for display states, but not for action states. This feature is useful when states may require information from predecessor states.

4.7.2 Error Handling

The State Manager supports the integration of data entry and error pages. It does so by passing an error flag and environment variables back to a display state. The page associated with the display state may then be written to check for an error flag, and if it was set, retrieve data from environment variables rather than database beans.

4.7.3 Form Validation

The State Manager supports binding of form input to a pipeline component to a validator class. This allows for very clean and powerful validation of form input. Classes used to provide this functionality are outlined in the Appendix.

4.7.4 Parameterized Pipeline Components

The State Manager supports associating key-value pairs with action states. This allows parameterization of application logic.

4.7.5 Transaction Management

The State Manager provides basic transaction management across pipeline components. Individual pipeline components may throw an `AbortTransactionException` if their success is vital to the transaction. Each component in the transaction is then given an opportunity to rollback the transaction.

4.7.6 Database Connection Management

While developers are encouraged to use EJBs and services rather than querying databases directly, the State Manager does provide support for database connectivity via JDBC. The `RequestContext` class has two functions for retrieving a database connection: one for a shared connection and one for an exclusive connection. The rules for obtaining a connection during a request are as follows.

Initially, no database connection is held in the `RequestContext`.

When a shared connection is requested, if the `RequestContext` does not have a connection, a shared connection will be retrieved and held in the `RequestContext`. If the `RequestContext` already has a connection (either shared or exclusive), that connection will be returned.

When an exclusive connection is requested, if the `RequestContext` does not have a connection, an exclusive connection will be retrieved and held in the `RequestContext`. If the `RequestContext` has an exclusive connection, that connection will be returned. If the `RequestContext` has a shared connection, an exclusive connection will be retrieved and stored in the `RequestContext`, replacing the shared connection.

The `RequestContext` will close and remove any database connection when the request is complete. Therefore the `RequestContext` should not close the connection it retrieves.

4.7.7 Additional Classes

Various additional classes used by the State Manager are outlined in the Appendix.

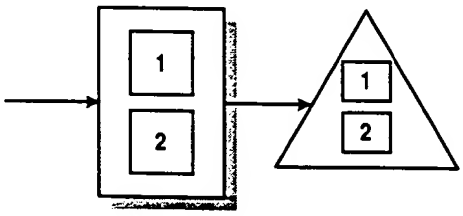
5 Usage Scenarios

This section explores the kinds of things that can be done with various embodiments of the State Manager. It examines some various problem scenarios and how the State Manager may be applied to solve these problems. In the process, various advanced features of the State Manager are introduced.

5.1 Overloaded Form Processor

This example looks at a common scenario when doing server-side processing of form data. Consider the case where a single form returns two addresses: one for receiving payments and one for the physical store location. Hitting submit on this form returns control to the server, which retrieves form variables from the page and updates the addresses in the database before redirecting to another page.

The following diagram depicts this situation. One HTML page has essentially two separate sets of information in its form. It submits to the server, which then execute two address updates using a common processor.

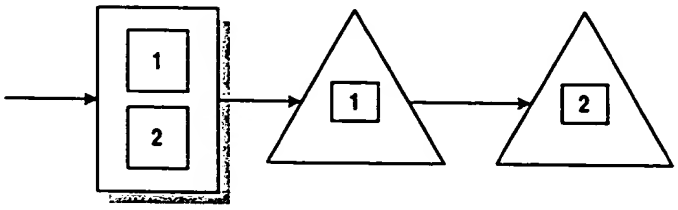


Overloaded Form Processor

There are obvious problems with this situation. What happens if a new reseller decides it wants to collect and update the payment and physical addresses on separate forms? The obvious solution would be to write two new form-processing scripts and two new HTML pages to get this functionality. This is tedious, error-prone, and makes maintenance and upgrades more difficult.

The State Manager allows this part of the application to be redesigned. Rather than having a single form processor for all the information on the form, we will write two individual pipeline components to perform the two distinct database updates. Once we have divided the form processor code into two parts, we have a lot more flexibility.

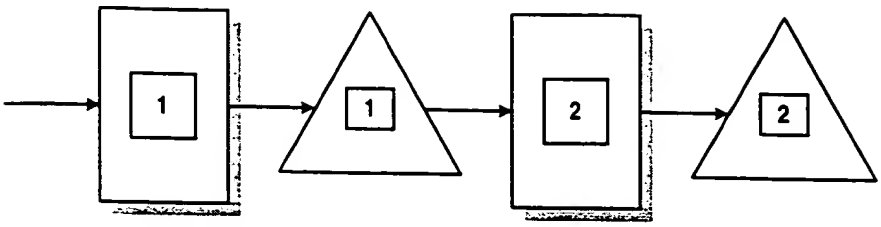
One option is to keep the same user interface. One HTML page has the form parameters needed for the two components. Submitting the form on the HTML page moves to the first action state, which performs its database update and then sends execution to the second action state.



Separate Components

Using the same code components, we can now easily meet our reseller's request to divide the UI into separate forms.

The following diagram depicts form 1 (payment address) posting to the pipeline component for updating the payment address. After this, the user is directed to the second form (physical address). A submit from that form then calls the pipeline component for updating the physical address.



Separate Components, Separate Forms

The State Manager enables this is a new level of flexibility. This example demonstrates how a more modular component-based system allows for the user interfaces to be more modular as well. Specifically, HTML forms that drive specific updates may be composed into a single page, or pulled apart into separate pages, while still sharing common application logic.

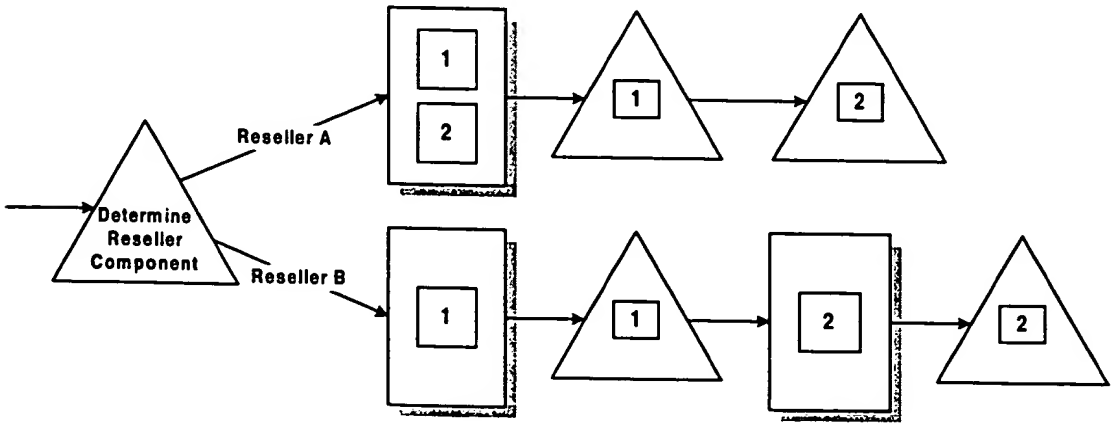
5.2 Application Needs To Change Based On Some Information Not Embedded In State Diagram

The State Manager allows for application flow to be affected by internal logic. This is done by allowing pipeline components to return an event, which may then be used to drive the next state. While most pipeline components will probably either succeed or fail (and go to the corresponding succeed or failure state), this is not a constraint. Pipeline components may also be used to direct the State Manager by returning any kind of event.

Consider the example of two resellers within the same deployment of an application requiring the application to behave differently. Using the address form situation from the first example Reseller A wants a single form to drive both database updates. The Reseller B wants two distinct forms.

The State Manager allows for a straightforward solution to this situation. First we write a new pipeline component whose output event is the Reseller ID of the current user. We then use this component to drive the application to either of the alternatives.

The following diagram depicts this situation. The "Determine Reseller Component" would be the first action state to which the user is sent. The output of the component, Reseller ID, drives the application to the appropriate variant. Each variant uses common application logic, with changes constrained to the presentation layer.



Reseller Driving Application Variations

Determining the reseller is just one example of how the application may be dynamically controlled. Logical "decision" components may be written any time that we need to change the application dynamically.

5.3 Duplicate Request Handling

Duplicate requests can be a major problem for web applications. The State Manager provides elegant duplicate request handling in a transparent way.

5.3.1 What Is A Duplicate Request?

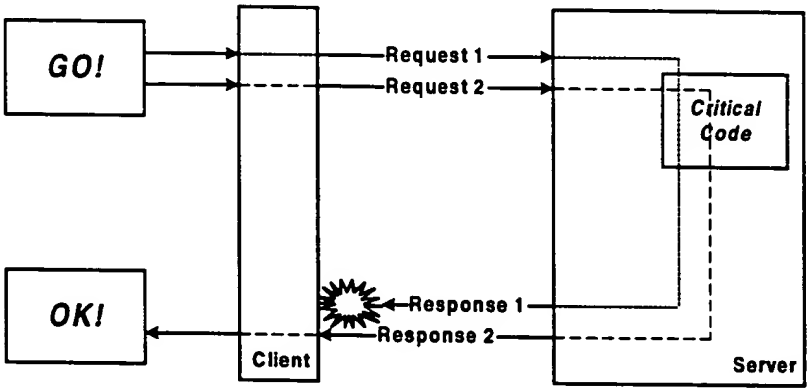
A duplicate request may occur when a user clicks on a web page's submit button more than once before the server returns a response. This sends the same request to the browser multiple times. The client ignores the response of the first request and displays the response of the last request.

Duplicate requests can be quite common. Anytime a user clicks a button and does not perceive any change, he may grow impatient and click the button again. With web-based applications, this may happen quite often.

5.3.2 Why Are Duplicate Requests Bad?

Typically server processes are not programmed to identify a duplicate request. This means the server will processes both requests, perhaps attempting to perform some critical action (such as charging a credit card) more than once.

Here is a picture of a duplicate request in terms of client-server interaction:



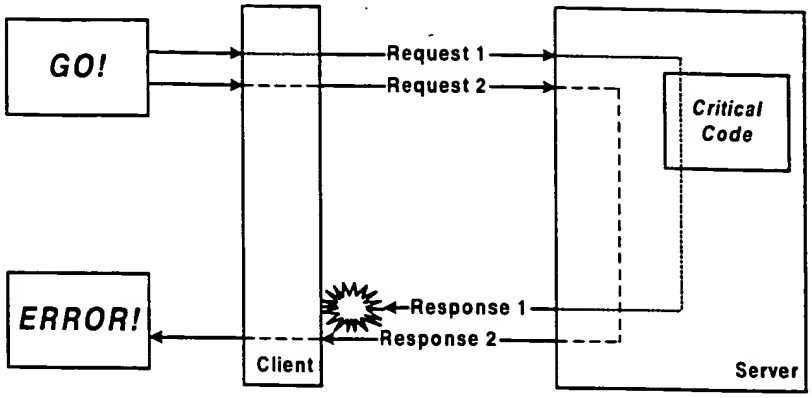
Duplicate Request: No Handling

Request 1 and Request 2 are identical and occur within milliseconds of each other- the result of a user double clicking a submit button on a form. The server (the dotted lines inside the server) handles both requests, and both requests generate responses. The response for Request 1 does not make it to the client because the client is not listening for it. Instead, the client listens only for the response to Request 2, which it does receive. Both requests executed the critical code, which might make database tables inconsistent, charge a credit card twice, or cause other damaging side effects.

5.3.3 Typical Duplicate Request Handling

The above scenario (no duplicate request handling) is commonplace in web applications. To address these problems, some web application developers write specific duplicate request handling functionality into their server code. Typically this code checks for two requests made by the same user within a very small amount of time. If a duplicate request is detected according to these parameters, the server stops processing the duplicate request and returns an error message to the user.

Here is a diagram of typical duplicate request handling with more sophisticated web applications:



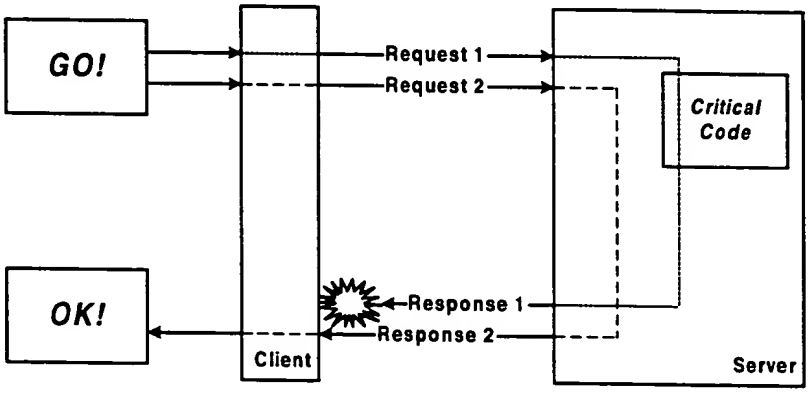
Duplicate Request: Common Handling

In this case, the server has been safeguarded from the harm of a duplicate request because the duplicate request does not reach the critical code. It is, in effect, turned away at the door.

This solution has its own issues though. The client receives only an error message informing him that he made a duplicate request. The server must return an error message, rather than the proper response, because the first and second requests were processed in two separate, disconnected threads. The second thread has no way of returning to the client the response of the first request, which would be ideal, because the client ignores the response from the first request.

5.3.4 Kinzan State Manager Duplicate Request Handling

The State Machine architecture allows for more elegant and natural duplicate request handling. Because the State Manager maintains context information at the page and response levels, it can accurately detect duplicate requests. Then, when a duplicate request is detected, it can return the proper output to the client.



Duplicate Request: Handled Properly By The State Machine

The State Manager can very accurately determine if a request is a duplicate. This is because a `RequestContext` object is saved with each request. The `RequestContext` contains all of the form variables and values that belong to the request. A new request may be compared against the previous one, value by value.

This is a great advantage over many duplicate request determination algorithms, which simply look for two requests coming from the same user within a very small amount of time (usually milliseconds), and which give no consideration to the actual request values. This type of algorithm is only useful with the accidental "nervous twitch" double-click, and not with the more common situation where a user is uncertain if the button click was accepted and decides to click it again.

How does the State Manager return the proper output to the client? The State Manager never actually produces any output itself. Instead, it redirects or forwards the user to a new location any time a display state is reached. When a duplicate request is detected, the second request may find the resulting display state of the first request (the one the user really wants to see) and redirect the client to that page.

What if the first request is not finished when the second request tries to find the result display state? In this case the State Manager redirects the client to a temporary "Please Wait" page. This page will automatically refresh, checking the server again to see if the first request has finished. If it has, the State Machine will return the result state, otherwise it will return the "Please Wait" page again.

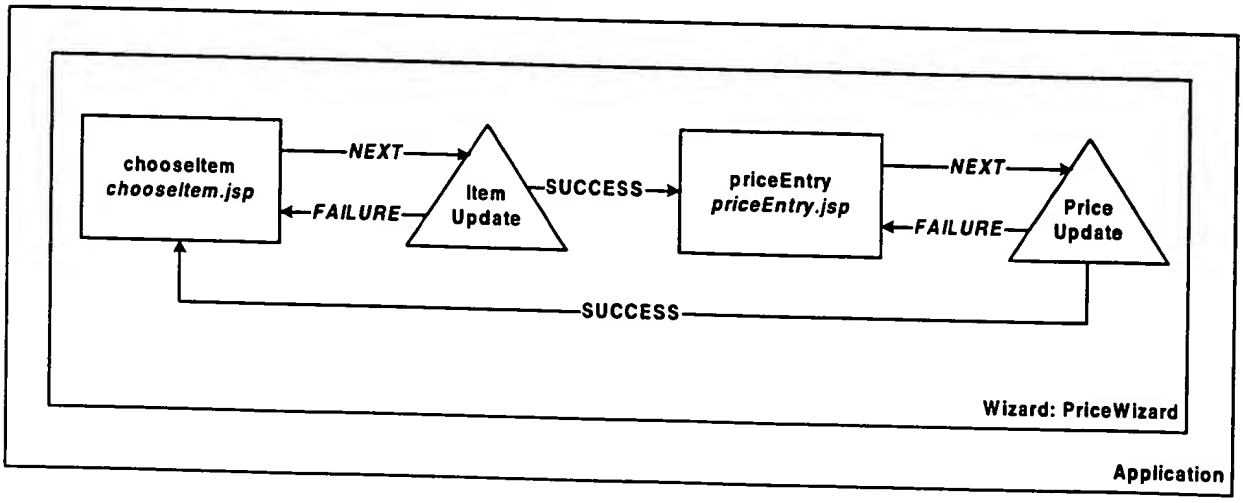
The result is a very elegant and natural handling of duplicate requests. The server and user are safeguarded from the harm of duplicate requests, the user gets warned when he double-clicks (potentially avoiding future duplicate requests), and the user gets the appropriate response when the original request finishes.

6 Using The State Manager

The following section illustrates an exemplary implementation of an embodiment of a state manager.

6.1 Example State Diagram XML file

State Diagrams are defined and loaded into the State Manager using wizards, which are XML files. The DTD for these files is listed in the Appendix. Consider the example of a simple application that allows users to enter the price for different items.



This wizard would be represented by the following XML file:

```
1  <?xml version="1.0"?>
2  <!DOCTYPE wizard SYSTEM "http://www.kinzan.net/dtd/wizard.dtd">
3
4  <wizard name="PriceWizard">
5     <displayState name="chooseItem" forward="/templates/tax/chooseItem.jsp">
6        <transition event="NEXT" state="xItemUpdate"/>
7     </displayState >
8     <displayState name="priceEntry" forward="/templates/tax/priceEntry.jsp">
9        <transition event="NEXT" state="xPriceUpdate"/>
10    </displayState>
11    <actionState name="xItemUpdate" component="test.component.ItemUpdate">
12       <transition event="SUCCESS" state="priceEntry"/>
13       <transition event="FAILURE" state="chooseItem"/>
```

```
14    </actionState>
15    <actionState name="PRICE_UPDATE" component="test.component.PriceUpdate">
16          <transition event="SUCCESS" state="chooseItem"/>
17       <transition event="FAILURE" state="priceEntry">
18          <property name="errorMessage" value="Price entry failed"/>
19       </transition>
20    </actionState>
21 </wizard>
```

4 Defines the wizard name within the domain.

5-7 Defines and configures the `chooseItem` display state to display `chooseItem.jsp` and forward control to the `xItemUpdate` action state after a `NEXT` event.

8-10 Defines and configures the `priceEntry` display state to display `priceEntry.jsp` and forward control to the `xPriceUpdate` action state after a `NEXT` event.

11-14 Defines and configures the `xItemUpdate` action state to use `ItemUpdate` as its pipeline component. Control is passed to the `priceEntry` display state after a `SUCCESS` event, or back to the `chooseItem` display state after a `FAILURE` event.

15-20 Defines and configures the `xPriceUpdate` action state to use `PriceUpdate` as its pipeline component. Control is passed to the `chooseItem` display state after a `SUCCESS` event, or back to the `priceEntry` display state after a `FAILURE` event. If there is a failure event, the "`errorMessage`" property will be set to "Price entry failed" in the request context before a transition is made to the `priceEntry` display state.

6.2 Example Pipeline Component

Pipeline components are Java classes that implement the `PipelineComponent` interface. The following example is a trivial pipeline component that compares a name against the name property it is configured with.

```
1  public class CheckNameComponent extends BasicComponent
2  {
3     private String iName = null;

4     public void init( Properties props )
5     {
6        iName = props.getProperty( "NAME" );
7     }

8     public String processEvent( RequestContext context )
         throws InvalidParameterException
9     {
10       String name = context.getProperty( "NAME" );

11       if ( name == null )
12       {
13          throw new InvalidParameterException( "You must enter a name." );
14       }
15       if ( name.equals( iName ) )
16       {
17          System.out.println( "You entered the right name!" );
18          return SUCCESS;
19       }

20       System.out.println( "The name you entered, " + name + ", was wrong." );
21       return FAILURE;
```

```
22     }
23 }
```

3-7 Declare a private variable `iName` and set it with the `NAME` property associated with the pipeline component.

8 `processEvent` is called by the State Manager, which passes the request context to the pipeline component.

10 Retrieve the form variable for `NAME` from the request context.

11-14 If the form name is null, throw an exception.

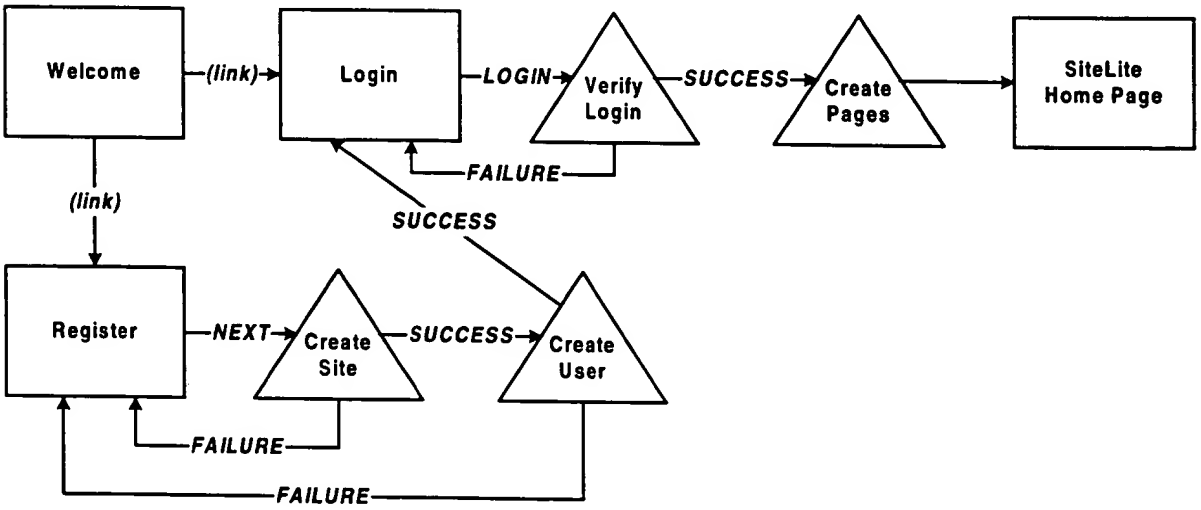
15-19 Return a `SUCCESS` event if the form name is the same as the property name.

20-21 Return a `FAILURE` event if the form name is not the same as the property name.

6.3 Login Example

6.3.1 State Diagram

Consider the following state diagram, which could be used to manage the login and new user creation portions of an application called "SiteLite"



The application begins on a welcome page that offers users a link (application navigation) to either login or register as a new user. Once a new user account in created and the user has successfully logged in, control is passed to the home page for the application.

The XML file that defines this state diagram is as follows:

```
1  <?xml version="1.0"?>
2  <!DOCTYPE wizard SYSTEM "http://www.kinzan.net/dtd/wizard.dtd">
3
4  <wizard name="login">
5     <displayState name="login" ksp="login">
6        <transition event="LOGIN" state="xVerifyLogin"/>
7     </displayState>

8     <displayState name="register" ksp="register">
9        <transition event="NEXT" state="xCreateSite"/>
10    </displayState>
```

```
11      <displayState name="siteliteHomepage" forward="/frameset.html"/>

12    <actionState name="xVerifyLogin" component="sitelite.component.VerifyLogin">
13       <property name="parameter.1" value="USERNAME"/>
14       <property name="parameter.1.validator.1"
              value="net.kinzan.state.pipeline.validators.SimpleStringValidator"/>
15       <property name="parameter.1.validator.1.maxLength" value="10"/>
16       <property name="parameter.1.validator.1.minLength" value="5"/>

17       <transition event="SUCCESS" state="xCreatePages"/>
18       <transition event="FAILURE" state="login"/>
19    </actionState>

20    <actionState name="xCreateSite" component="sitelite.component.CreateSite">
21       <transition event="SUCCESS" state="xCreateUser"/>
22       <transition event="FAILURE" state="register"/>
23    </actionState>

24    <actionState name="xCreateUser" component="sitelite.component.CreateUser">
25       <transition event="SUCCESS" state="login"/>
26       <transition event="FAILURE" state="register"/>
27    </actionState>

28    <actionState name="xCreatePages"
                   component="sitelite.component.CreateSitePages">
29       <transition event="SUCCESS" state="siteliteHomepage"/>
30       <transition event="FAILURE" state="siteliteHomepage"/>
31    </actionState>

32 </wizard>
```

5-7 Define display state for login page (in this case, a KSP; the Rendering Service is automatically invoked).

8-11 Define display states for Register (KSP) and SiteLite home page (standard HTML). Note that the display state for the SiteLite home page forwards directly to an HTML page instead of a KSP.

12 Define `xVerifyLogin` action state to be associated with `VerifyLogin` pipeline component

13 Establish a property for the `xVerifyLogin` action state named `USERNAME`

14 Associate a `SimpleStringValidator` with the `USERNAME` property

15-16 Configure the `SimpleStringValidator` to require a `USERNAME` with at least 5 characters but no more than 10 characters

20-31 Define the other action states (and associated pipeline components) in the state diagram

The rest of this section will demonstrate how the Rendering Service and the State Manager work together by focusing on the `login` display state and the `xVerifyLogin` action state.

6.3.2 `login.ksp`

The `login` display state is associated with the `login` KSP of the `sitelite` site. The following is the KSP for the login page, `login.ksp`:

```
1  <?xml version="1.0"?>
2  <!DOCTYPE ksp SYSTEM "http://www.kinzan.net/dtd/ksp.dtd" >
3
```

```
4  <ksp>
5     <title>SiteLite Login</title>
6     <structure>
7        SiteLiteIntro
8     </structure>
9     <style>
10       Clean
11    </style>
12    <zoneList>
13       <zone name="MAIN">
14          <widget type="kco">
15             <property name="file" value="login.kco"/>
16          </widget>
17       </zone>
18    </zoneList>
19 </ksp>
```

5 Set the title of the page.

6-8 Configure the page to use the `SiteLiteIntro` structure (defined in the kapp file for the site and already loaded into the runtime environment).

9-11 Configure the page to use the `Clean` style (defined in the kapp file for the site and already loaded into the runtime environment).

12-18 Define the single zone on the page to include `login.kco` by using a widget of type kco.

Here is the `login.kco` asset referenced by the `login.ksp` file:

```
1  <%@ page import="net.kinzan.servlet.RequestContext,
                    net.kinzan.state.pipeline.InvalidParameterException"%>
2  <%RequestContext context =
      (RequestContext)request.getAttribute( "net.kinzan.requestContext" );%>

3  <font CLASS="title">Log In</font>

4  <%
5     String username = "";
6     String password = "";
7     Exception error = context.getError();
8     if ( error != null ) {

9        username = context.getProperty( "USERNAME" );
10       password = context.getProperty( "PASSWORD" );

11       %>
12       <BR><font CLASS="error">
13          <%= error.getMessage()%>
14       </font>
15 <% } %>

16 <div CLASS="form">
17 <form name="MAIN" action="/state" method="POST">
18    <input type=hidden name="net_kinzan_requestContextID"
            value="<%= context.getID()%>">

19    <table>
20       <tr>
21          <td><font>Username: </font></td>
```

```
22          <td><input type="TEXT" name="USERNAME" value="<%= username%>">
23           </td>
24        </tr>
25        <tr>
26           <td><font>Password: </font></td>
27           <td><input type="PASSWORD" name="PASSWORD" value="<%= password%>" >
28           </td>
29        </tr>
30     </table>
31     <br><br>
32     <input type="HIDDEN" name="net_kinzan_nextEvent" value="LOGIN">
33     <a href="javascript:document.MAIN.submit();">
34        <widget type="ref" name="nextbutton"/>
35     </a>
36 </form>
37 </div>

38 <script>
39    document.MAIN.USERNAME.focus();
40 </script>
```

2 Retrieve the request context for the page (recall, KCO's are JSPs which are style and widget aware).

3 Use the `title` class for the font style.

7 Retrieve the error message from the request context.

8-15 If there is an error message (from the `VerifyLogin` pipeline component in this case), retrieve the previously input username and passport from the request context and print the error message using the `error` font class (allows for common styling of all error messages on all pages).

17 Login form uses the State Manager as the `ACTION`.

18 A hidden form variable to identify the form (via the request context ID) to the State Manager.

19-31 Standard HTML form to input the username and password.

32 A hidden form variable to declare that submit from the form should result in a `LOGIN` event to trigger the next state in the wizard.

33-35 The submit button (in this case, a site-wide widget named `nextbutton`)

Note that error handling is embedded in the same display state as the original input. This is made possible because the request context is available to the display state and obviates the need for a special variant of the display state to manage error conditions.

6.3.3 `VerifyLogin.java`

The role of the `VerifyLogin` pipeline component is to verify the login and password entered by a user matches up to the appropriate entry in the LDAP directory for the site, and to communicate the correct information to other states in the application. It uses a localizable resource bundle to store the error message in case of an incorrect login.

The Java class is as follows:

```
1  package com.kinzan.example.sitelite.component;

2  import net.kinzan.state.pipeline.ComponentParameter;
```

```
3  import net.kinzan.state.pipeline.InvalidParameterException;
4  import net.kinzan.servlet.RequestContext;

5  import net.kinzan.state.pipeline.component.BasicLogComponent;

6  import net.kinzan.servlet.RequestContext;
7  import net.kinzan.servlet.AppContext;
8  import net.kinzan.servlet.KinzanApp;

9  import java.util.Locale;
10 import java.util.ResourceBundle;
11 import java.util.Properties;

12 import com.kinzan.example.sitelite.UserManagerService;

13 import javax.servlet.http.*;

14 import java.text.MessageFormat;

15 /**
16  * Authenticates a user based on username and password.
17  */
18 public class VerifyLogin
       extends BasicLogComponent
19 {
20    public VerifyLogin()
21    {
22       super();

23       ComponentParameter username = new ComponentParameter( "USERNAME" );
24       username.setNullOkay( false );

25       ComponentParameter password = new ComponentParameter( "PASSWORD" );
26       password.setNullOkay( false );

27       iInputParameters.addComponentParameter( username );
28       iInputParameters.addComponentParameter( password );
29    }

30    public String processEvent( RequestContext context )
31    {
32       boolean validLogin = false;
33       if ( iTrace.isLogging )
34          iTrace.entry( TYPE_PUBLIC, iClassName, "processEvent" );

35       // Get locale-specific messages.
36       Locale locale = context.getServletRequest().getLocale();
37       ResourceBundle res =
            ResourceBundle.getBundle( VerifyLogin.class.getName(), locale );

38       String username = context.getProperty( "USERNAME" );
39       String password = context.getProperty( "PASSWORD" );

40       if ( iTrace.isLogging )
41       {
42          iTrace.text( TYPE_SVC, iClassName, "processEvent",
                         "***** The username is '" + username + "'. *****" );
43          iTrace.text( TYPE_SVC, iClassName, "processEvent",
                         "***** The password is '" + password + "'. *****" );
44       }

45       try
46       {
```

```
47          // Retrieve the UserManagerService that will create the user.
48          UserManagerService userManager =
        (UserManagerService)context.application.getService( "UserManagerService" );

49          // Create the user.
50          validLogin = userManager.authenticateUser( username, password );
51          if ( validLogin )
52          {
53             Properties userProps = userManager.getUser( username );

54             String sitename = userProps.getProperty( "cn" );

55             HttpSession session = context.getSession();
56             session.setAttribute( "sitelite.username", username );
57             session.setAttribute( "sitelite.password", password );
58             session.setAttribute( "sitelite.sitename", sitename );
59          }
60       }
61       catch ( Exception e )
62       {

63          e.printStackTrace();
64       }


65       if ( !validLogin )
66       {
67          HttpSession session = context.getSession();
68          session.removeAttribute( "sitelite.username" );
69          session.removeAttribute( "sitelite.password" );
70       }


71       if ( iTrace.isLogging )
72          iTrace.exit( TYPE_PUBLIC, iClassName, "processEvent", validLogin );

73       if ( !validLogin )
74          throw new InvalidParameterException(
              res.getString( "INVALID_USERNAME_PASSWORD" ) );

75       return validLogin ? SUCCESS : FAILURE;
76    }

}
```

23-24 Set up a component parameter called `user name`, associate it with the form variable `USERNAME`, and configure it to throw an exception if null.

25-28 Repeat for a component parameter for `password` and add both to the component list for the pipeline component.

37 Retrieve the resource bundle associated with the pipeline component.

25-26 Retrieve the form variables `USERNAME` and `PASSWORD` from the request context (passed to the pipeline component by the State Manager).

48-50 Use the User Manager service to connect with the LDAP server and pass the username and password for authentication.

51-59 If valid login, store username and password in the session for the user

65-70 If invalid login, make sure username and password is not in the session

73-74 If invalid login, throw an invalid parameter exception with the `INVALID_USERNAME_PASSWORD` string from the resource bundle associated with the pipeline component

75 Return a `SUCCESS` event if a valid login; otherwise return a `FAILURE` event

The resource bundle for this pipeline component is straightforward. By extracting the exception message into a resource bundle, it may be uniquely configured for different implementations. More importantly, it may be easily localized to different languages using built in Java localization support.

The English version of `VerifyLogin_res.properties` follows:

```
1  # messages for sitelite.component.VerifyLogin

2  INVALID_USERNAME_PASSWORD=Invalid username/password combination.
```

2 Associate the string "Invalid username/password combination" with the `INVALID_USERNAME_PASSWORD` property

7 Additional Definitions and Details

The following section provides further implementation-specific or embodiment-specific details which are illustrative and exemplary in nature.

7.1 Additional Classes

`ComponentLoader`

The `ComponentLoader` interface describes a class whose sole purpose is to find `PipelineComponent` objects. The State Manager uses a `ComponentLoader` to instantiate the `PipelineComponents` described in the `ActionState` objects it retrieves.

```
/**
 * Retrieves a PipelineComponent object using an ActionState
 * as its key.  The component name is retrieved from the
 * ActionState.getPipelineComponent() method.
 */
PipelineComponent getComponent( ActionState state );

/**
 * Retrieves a PipelineComponent object using a String key
 * and a set of initialization properties.
 */
PipelineComponent getComponent( String comp, Properties props );
```

The State Manager has a default `ComponentLoader` (the `ClassForNameLoader`). It assumes that the component String name is the fully qualified class name of the `PipelineComponent` and uses `Class.forName( name )` to load the class and then `newInstance()` to return a new instance of the object.

`RequestContext`

The `RequestContext` class provides access to all parameters/session/request information. It has separate methods to get the session, request, and input parameters:

```
void setID( String id );  // set the context ID
String getID(); // get the context ID
```

```
HttpSession getSession(); // return session
HttpServletRequest getServletRequest(); // return servlet request object
HttpServletResponse getServletResponse(); // return servlet response object .

String getProperty( String key ); // get String input property
Object get( String key ); // get non-String input property

void setProperty( String key, String value ); // set String value
void set( String key, Object value ); // set Object value

void clear(); // clear all the values in this context

boolean isDuplicate(); // is this a duplicate request?

void startRequest(); // called at the start of a request
void endRequest( boolean commit );    // Either endRequest() or abortRequest()

Connection getExclusiveConnection (); // get an exclusive database connection
Connection getSharedConnection (); // get a shared database connection

void setCurrentState( State state ); // set the current state
State getCurrentState(); // get the current state

void setError( Exception e ); // set the current error
Exception getError(); // get the current error
```

The `RequestContext` is responsible for containing the input of each component in a pipeline, and for adding the output of each component to itself before moving on. A `RequestContext` is created at the beginning of a request, and is then associated with the response.

`ComponentParameter`

The `ComponentParameter` class encapsulates each input or output parameter for a pipeline component.

Each `ComponentParameter` contains:

```
Class objectType;
String internalName;
String externalName; (null)
ParameterValidator validator; (null )
```

Each parameter has an internal name and an external name. The internal name is how the component will reference the parameter within its code. The external name is how the State Manager will match the output of one component to the input of another. The internal and external names will match unless the external name is explicitly overridden. This allows for complicated scenarios where constant parameter names would be a difficult constraint.

A simple example is a component that is designed to process an Address object to issue a mailing. But let's say that when we need to call it, we have both a "SHIPPING_ADDRESS" and a "BILLING_ADDRESS" in the pipeline. The separation of external from internal parameter names allows the component to reference the proper address.

Internal names will be hard coded into the `PipelineComponent`. External names will be set when the pipeline component is created.

`ComponentParameterList`

The `ComponentParameterList` class is a container for `ComponentParameter` objects. It is able to enumerate the components by internal/external key, and allow for retrieval of the components by those keys.

```
Enumeration getInternalKeys()
```

```
Enumeration getExternalKeys()
ComponentParameter getComponentParameter( String internalKey );
ComponentParameter getComponentParameterExternal( String externalKey );
String getExternalKey( String internalKey );
```

ParameterValidator

The `ParameterValidator` interface defines a class that is responsible for validating a parameter. It has one method, `validate()`, which throws an exception if its input object does not meet its requirements. Different validators may be developed to check for very specific things, or they may be parameterized to be reusable in a generic fashion.

One example might be a `SimpleStringValidator`, which can check for null, and for maximum and minimum lengths. A more advanced string validator might use a regular expression check to make sure an email address is in the form "someone@somewhere.com".

The interface is as follows:

```
void validate( Object obj ) throws InvalidParameterException
```

InvalidParameterException

The `InvalidParameterException` is thrown by the `ParameterValidator` classes. It may contain useful information about the parameter that failed and why.

```
public String getName() // The external (form) name of the parameter
public String getParameterClass() // The name of the object type of
                                  // the invalid parameter
```

In addition to the exception message, an `InvalidParameterException` contains the name of the parameter (Form variable name), which allows developers to use client-side JavaScript to focus the cursor on the invalid parameter.

7.2 Document Type Definition (DTD) For Wizard Files

The DTD for wizard files in one embodiment is as follows:

```
<?xml encoding="US-ASCII"?>

<!ELEMENT wizard (actionState*,displayState*)*>
<!ATTLIST wizard
          name CDATA #REQUIRED
          startState CDATA #IMPLIED>

<!ELEMENT displayState (transition*,property*)*>
<!ATTLIST displayState
          name CDATA #REQUIRED
          forward CDATA #IMPLIED
          redirect CDATA #IMPLIED
          ksp CDATA #IMPLIED>

<!ELEMENT actionState (transition*,property*)*>
<!ATTLIST actionState
          name CDATA #REQUIRED
          component CDATA #REQUIRED>

<!ELEMENT property EMPTY>
<!ATTLIST property
          name CDATA #REQUIRED
          value CDATA #REQUIRED>

<!ELEMENT transition EMPTY>
<!ATTLIST transition
          event CDATA #REQUIRED
```

```
state CDATA #REQUIRED
wizard CDATA #IMPLIED
clearContext CDATA #IMPLIED>
```

A utility class is available to read XML files conforming to this DTD and create the corresponding objects within the State Manager. This class has the following public methods:

```
// Read a single file and return a Domain defined by the file's XML.
public Domain GetDomainFromXMLFile( File xmlFile );

// Compose a Domain from multiple definition files.
public Domain GetDomainFromXMLFiles( File xmlDir );
```

7.3 Deployment

A deployment of the State Manager involves the State Manager JARs, the various pipeline components and state diagrams required for a domain, and access to the appropriate JSP/KSP files and/or HTML pages for display states.

7.3.1 State Manager JARs

The State Manager code is the set of class files that make up the State Manager Framework. Deploying these involves placing a JAR of this code in the `classpath` of the Java Virtual Machine (JVM) of the server. Updates to this code would require a new JAR to be deployed. The State Manager is part of the Fountainhead distribution is not modifiable by external developers.

7.3.2 Pipeline Components

Any new or updated pipeline components must be accessible to the `ComponentLoader`, which is used by the State Manager to find and load pipeline components. The default `ComponentLoader`, `ClassForNameLoader`, expects to find the class definition and create a new instance of the class using `Class.forName()` and `Class.newInstance()`. For pipeline components that are loaded using this loader, deploying the pipeline components will mean placing the class files (or jar of class files) in the `classpath` of the JVM.

When pipeline components are deployed to a production server, they are registered with a lookup server so as to be visible to the `ComponentLoader`.

7.3.3 State Diagrams

Wizards (a set of states, events, and pointers to pipeline components and pages) is stored in an XML file. These files are loaded at runtime into the system to generate the `Wizard` object. Alternatively, the developer may choose to have the `Wizard` object cached in private database tables to obtain greater speed and reliability. In this case, deploying a new domain will involve using a utility class to load the domain from the XML file and persist the cached `Wizard` in the private database.

7.3.4 JSP/KSP Files and HTML Pages

The display states described in the wizard refer to either a KSP file (via the `ksp` attribute) or any http addressable page (via the `forward` attribute). New or updated files need to be deployed to the document root of the web site to be visible to the State Manager.

In the preceding description, the method and apparatus of the present invention is described with reference to specific exemplary embodiments thereof. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the present invention. In particular, the separate blocks of the various block diagrams represent functional blocks of methods or apparatuses and are not necessarily indicative of physical or logical separations or of an order of operation inherent in the spirit and scope of the present invention. For example, the various blocks of some figures may be integrated into components, or may be subdivided into components. Moreover, the various blocks of other figures represent portions of a method which, in some embodiments, may be reordered or may be organized in parallel rather than in a linear or step-wise fashion. The present specification and figures are accordingly to be regarded as illustrative rather than restrictive.

CLAIMS

What is claimed is:

1. A system comprising:

a system processor;

a set of widgets accessible by the system processor; and

a state machine accessible by the system processor.

2. A method comprising:

processing a set of web pages;

instantiating a set of widgets responsive to the processing; and

manipulating a state machine responsive to the processing.